

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



精通Git的使用技巧，掌握高级分布式版本控制特性，高效提升团队协作生产效率

# Git高手之路

Mastering Git

[ 波兰 ] 雅各布·纳热布斯基 ( Jakub Narębski ) 著  
邓世超 译



# Git高手之路

[波兰] 雅各布·纳热布斯基 (Jakub Narębski) 著  
邓世超 译

人民邮电出版社  
北京



## 图书在版编目 (C I P) 数据

Git高手之路 / (波) 雅各布·纳热布斯基著 ; 邓世超译. — 北京 : 人民邮电出版社, 2018. 4  
ISBN 978-7-115-47850-4

I. ①G… II. ①雅… ②邓… III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆CIP数据核字(2018)第019195号

## 版 权 声 明

Copyright ©2016 Packt Publishing. First published in the English language under the title *Mastering Git*.  
All rights reserved.

本书由英国 Packt Publishing 公司授权人民邮电出版社出版。未经出版者书面许可, 对本书的任何部分不得以任何方式或任何手段复制和传播。

版权所有, 侵权必究。

---

◆ 著 [波兰] 雅各布·纳热布斯基 (Jakub Narębski)  
译 邓世超  
责任编辑 胡俊英  
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
三河市潮河印业有限公司印刷

◆ 开本: 800×1000 1/16  
印张: 23.5  
字数: 465 千字 2018 年 4 月第 1 版  
印数: 1-2 400 册 2018 年 4 月河北第 1 次印刷

著作权合同登记号 图字: 01-2016-8084 号

---

定价: 89.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316  
反盗版热线: (010)81055315

# 内容提要

Git 是一款免费、开源的分布式版本控制系统，可以对或大或小的项目进行高效的版本管理。时至今日，Git 已经在项目开发领域发挥着重要作用，并且得到了广泛的应用。

本书旨在帮助读者深入理解 Git 架构，以及其内部的理念、行为和最佳实践。全书共分为 12 章，从基础知识讲起，陆续介绍了项目历史管理、使用 Git 进行程序开发、工作区管理、Git 协作开发、分支应用进阶、集成变更、历史记录管理、子项目管理、Git 的定制和扩展、Git 日常管理、Git 最佳实践等内容。

本书面向所有的 Git 用户，全面细致地向读者介绍有关 Git 的各项实用技巧，充分发掘它的潜力，更好地实现项目版本管理。

Jakub Narebski 自 Git 诞生之初就参与了 Git 的开发工作。他是 gitweb 子系统（Git 原始 Web 界面）的主要贡献者之一，是非官方的 gitweb 维护者。他创造、发布并分析了 2007 年至 2012 年的年度 Git 用户调查。您可以在 Git Wiki 上找到对这些调查的分析内容。他经常在技术问答网站 StackOverflow 上和他人分享自己的技术专长。

他是 Eric Sink 的 *Version Control by Example* 一书的审校者之一，这也是他在 Git 领域占有一席之地原因。

他是波兰托伦哥白尼大学数学和计算机科学系的助理教授。他选择使用 Git 作为个人和专业工作的版本控制系统，将其作为课程作业的一部分讲授给数学和计算机科学系的学生。

## 审稿人简介

Markus Maiwald 是一名互联网服务提供商，商业网站推广员和域名提供商。例如，他为客户提供一站式白色标签解决方案（从注册域名到部署 Web 服务器）。

因此，他的口号是：我们的系统，您的生意。

在专业领域，他是一名顾问和系统管理员，拥有超过 15 年的 Linux 经验。他喜欢构建高性能的服务器系统，还开发出了不少可重用、高安全性的标准系统。

作为一名真正的 Webworker 2.0，他的客户遍及全球，其范围从欧洲的一家保险公司到泰国的 Web 开发工作室。

这也是他热衷于本书相关工作的主要原因。作为一名伟大的团队协作成员，他在国际团队合作方面拥有丰富的经验，他在工作中引进了 Git 这样能够大幅度提高生产力的工具。

---

必须感谢来自 Packt 出版社的项目协调人 Bijal Patel，我得到了他的大力支持，并且度过了愉快的时光。

还要感谢 Sarah 在我完成此项目过程中给予的耐心和鼓励。

---



# 前言

本书是经过精心设计的，旨在帮助读者深入理解 Git 架构，以及其内部理念、行为和最佳实践。

本书以使用 Git 进行协作开发的简单项目示例入手，使得读者能够了解基本的 Git 操作和概念。然后，随着阅读的深入，后续的章节将会阐述不同领域的具体应用细节：从源代码历史版本管理，再到管理用户自己的工作成果，然后是和其他开发人员协作。版本控制主题应用伴随着 Git 架构和行为的细节讲解。

本书还有助于读者增强检查和浏览项目历史记录、创建和管理工作成果、中心式和分布式工作流中配置版本库和分支方便协作开发、集成来自其他开发人员的工作成果、自定义和扩展 Git，以及恢复版本库数据等能力。通过了解 Git 高级应用技巧和内部工作机制，读者将会深入理解 Git 的行为，使得用户可以自定义和扩展现有的工具和脚本，甚至编写符合自己需要的功能。

## 本书概要

第 1 章“Git 应用入门”向读者提供了 Git 在版本控制方面的基本应用介绍。本章重点放在了技术应用方面，在一个开发示例中展示和说明了基本的版本控制操作和两名开发人员之间的协作开发。

第 2 章“项目历史管理”引入了修订的有向无环图（DAG）概念，并且解释了该概念和 Git 分支、标签和当前分支的关系。读者还会学习到如何查询、过滤和查看项目历史中的修订区间，如何使用不同条件查找修订记录。

第3章“使用 Git 进行程序开发”讲述了如何创建和添加这类历史记录。读者将会学习如何创建新修订以及展开新的开发工作。本章引入了提交暂存区的概念（索引），以及解释了如何查看和读取工作目录、索引和当前修订版本之间的差异。

第4章“工作区管理”的重点是解释如何为准备创建新提交而管理工作目录，同时还向读者介绍管理文件目录的细节。其中也包括需要特别处理的文件类型，引入了忽略文件和文件属性等概念。

第5章“Git 协作开发”对多种协作模式进行了鸟瞰式的介绍，展示了不同中心式和分布式工作流之间的差异，它主要聚焦于协作开发过程中版本库层面的交互。这里读者还会学习到信任链的概念，以及如何使用签名标签、签名合并和签名提交。

第6章“分支应用进阶”深入介绍了分布式团队协作开发的细节，同时介绍了本地分支和远程版本库上的跟踪分支，以及如何同步分支和标签。读者将会学到多种分支技术，通过多种途径了解分支的类型和用途（包括主题分支工作流）。

第7章“集成变更”向读者介绍如何通过合并和变基操作合并来自不同开发流水线上的工作成果，同时还解释了不同类型的合并冲突，以及如何识别和解决它们。读者将会学习使用拣选提交拷贝变更，以及如何应用单个或批量补丁记录。

第8章“历史记录管理”解释了用户可能希望保持历史记录整洁的原因，以及应该这么做的时机和方法。这里读者将会找到如何重排、压缩和分割提交的详细步骤。本章还演示了如何恢复被重写过的历史记录，以及当用户无法重写历史记录时的解决方案：如何恢复提交，如何给它添加笔记，如何修改项目历史记录的视图。

第9章“子项目管理——构建活动框架”讲述和展示了通过不同方法在单个版本库的框架项目中连接不同项目，从通过将项目代码嵌入其他项目（子树）的强制性添加，到通过版本库（子模块）嵌套的方式连接两个项目。本章还介绍了处理大型版本库和大型文件的若干种解决方案。

第10章“Git 的定制和扩展”介绍了通过配置和扩展 Git 来满足用户的实际需要，还简要介绍了图形化接口。读者将会了解到如何配置命令行，使它更易用。本章解释了如何在 Git 中使用钩子完成自动化任务（重点是客户端钩子），例如如何让 Git 检查创建提交时，相关操作是否遵循了特定的代码规范的指导意见。

第11章“Git 日常管理”旨在帮助 Git 管理员提高日常管理工作的效率。它简要介绍了 Git 版本库服务的知识。这里读者将会学习如何使用服务端钩子处理日志记录、访问控制、强制性开发策略和其他任务。



第 12 章“Git 最佳实践”列出了一组通用的版本控制方法和特定于 Git 的建议和最佳实践。这些内容涵盖了工作目录管理，创建单个提交和一系列提交（pull 请求），提交附加变更和同行的代码审核等内容。

## 读者须知

为了运行本书涉及的示例和提供的命令，读者将需要安装 2.5.0 版本及以上的 Git 软件。Git 是开源的，并且兼容所有平台（例如 Linux、Windows 和 Mac OS X）。所有示例使用的都是 Git 文本化接口和 bash shell。

为了编译和运行第 1 章用到的示例程序（该程序主要用来演示基本的版本控制方法），读者需要安装 C 编译器和 make 程序。

## 目标读者

如果读者已经是一名 Git 用户，并且掌握了分支、合并、暂存和工作流等基本概念，那么本书是为你而写的。如果读者是 Git 的资深用户，本书可以帮助你了解 Git 的工作机制，充分挖掘它的潜力，并可以了解若干高级工具、技术和工作流。安装 Git 的基本知识和软件配置管理的概念是必需的。

## 排版约定

在本书中，读者将会接触到表达不同含义的若干种文本样式。这里将对它们逐一举例，并说明其代表的含义。

文本中的代码、命令以及选项、文件夹名、文件名、文件扩展名、路径名、分支和标签名、简易 URL 地址、用户输入、环境变量、配置选项和它们的参数值将会以如下格式表示：

“例如，`git log - foo` 命令中显式声明了历史路径 `foo`。”

此外，本书采用下列规范：`<file>` 表示用户输入（这里是一个文件名），`$HOME` 表示环境变量的值，路径名中的波浪字符表示用户的主目录（例如 `~/.gitignore`）。

代码块或配置文件片段以下列格式表示：

```
void init_rand(void)
```

```
{  
    srand(time(NULL));  
}
```

当代码块中需要引起读者的注意时（极个别情况），相关行会使用粗体表示：

```
void init_rand(void)  
{  
    srand(time(NULL));  
}
```

任何命令行的输入和输出采用如下格式表示：

```
carol@server ~$ mkdir -p /srv/git  
carol@server ~$ cd /srv/git  
carol@server /srv/git$ git init --bare random.git
```

新的术语和关键字会加粗表示。读者在屏幕上看到的词语（例如在菜单或者对话框中显示的文本），将会以如下格式表示：

“The default description that Git gives to a stash (**WIP on branch**).”



警告或需要特别注意的内容。



提示或者诀窍。

## 读者反馈

我们非常欢迎读者的反馈。告诉我们您觉得本书怎么样，以及您喜欢哪部分或不喜欢哪部分。有了读者的反馈，我们才能继续写出真正能让大家充分受益的作品。

如果您想反馈信息，很简单，请在异步社区网站与本书对应的页面发表评论，或者发私信给责任编辑。

如果您也是某个领域的专家，并且有兴趣编写或者合作出版一本书，请发送邮件至

contact@epubit.com.cn。

## 客户支持

很荣幸您是本书的读者，我们将为您提供物超所值的增值服务。

## 随书源码

您可以访问异步社区的网站，在本书对应的页面内下载配套文件。

## 随书插图

另外，我们也提供本书中快照和图表的彩色 PDF 格式文件，彩色图片有助于您理解输出结果的变化。您可以在异步社区网站上与本书对应的页面内下载到彩图文件。

## 勘误

虽然我们会全力确保书中内容的准确性，但错误仍在所难免。如果您在某本书中发现了错误（文字错误或代码错误），而且愿意向我们提交这些错误，我们将感激不尽。这样不仅可以消除其他读者的疑虑，也有助于改进后续版本。若您想提交所发现的错误，请访问异步社区网站，在本书对应的页面内提交勘误。一经核实，您所提交的勘误将在本书对应的勘误区域呈现给读者。

## 关于盗版

对所有媒体来说，互联网盗版都是一个棘手的问题，我们一直都很重视版权保护。如果您在互联网上发现我们公司出版物的任何非法复制品，请及时告知我们网址或网站名称，以便我们采取补救措施。

请通过 315@ptpress.com.cn 联系我们，并提供疑似盗版材料的链接信息。

感谢您帮助我们保护作者的权益，使我们能够为您提供更有价值的内容。





# 目录

第 1 章 Git 应用入门	1
1.1 版本控制与 Git	1
1.2 Git 简易示例	2
1.2.1 创建版本库	2
1.2.2 创建 Git 版本库	3
1.2.3 克隆版本库并添加 注释	4
1.2.4 发布修改	7
1.2.5 查看历史版本	7
1.2.6 重命名、移动文件	10
1.2.7 更新版本库（合并）	11
1.2.8 创建标签	12
1.2.9 解决合并冲突	14
1.2.10 添加和移除文件	17
1.2.11 撤销对单个文件的 修改	18
1.2.12 创建新分支	19
1.2.13 合并分支（无冲突）	20
1.2.14 撤销未发布的 合并	21
1.3 小结	22

第 2 章 项目历史管理	23
2.1 有向无环图	23
2.1.1 提交整个工作目录	25
2.1.2 分支和标签	26
2.1.3 分支点	28
2.1.4 合并提交	28
2.2 修订内部查询	28
2.2.1 HEAD——最新的 修订版本	29
2.2.2 分支和标签的引用	29
2.2.3 SHA-1 哈希码及其简化 标识符	30
2.2.4 父引用	32
2.2.5 反向父引用——git 的输出 信息描述	32
2.2.6 reflog 的简称	33
2.2.7 上游远程跟踪分支	34
2.2.8 根据提交信息查询修订	34
2.3 修订区间查询	35
2.3.1 单个修订内部查询	35
2.3.2 双点符号	35

2.3.3	多点符号——包含和排除 修订 .....	37	3.2.2	孤儿分支 .....	77
2.3.4	单个修订的修订区间 .....	38	3.2.3	分支的查询和切换 .....	77
2.3.5	三点符号 .....	38	3.2.4	分支列表 .....	80
2.4	历史记录查询 .....	40	3.2.5	分支的回退和复位 .....	80
2.4.1	限制修订数量 .....	40	3.2.6	分支的删除 .....	82
2.4.2	元数据查询 .....	40	3.2.7	分支的重命名 .....	83
2.4.3	修订内部变更查询 .....	43	3.3	小结 .....	83
2.4.4	变更类型查询 .....	44	第 4 章	工作区管理 .....	84
2.5	单个文件历史记录 .....	44	4.1	忽略文件 .....	85
2.5.1	路径约束 .....	45	4.1.1	将文件刻意标记为 不跟踪的 .....	86
2.5.2	历史简化 .....	46	4.1.2	确定忽略文件类型 .....	88
2.5.3	blame——查看文件 历史记录详情 .....	46	4.1.3	忽略文件列表 .....	89
2.6	使用 git bisect 命令查找 bug .....	48	4.1.4	忽略跟踪文件内的变更 .....	90
2.7	日志的查询和格式化输出 .....	50	4.2	文件属性 .....	91
2.7.1	预定义和用户自定义 输出格式 .....	51	4.2.1	配置 Diff 和 merge .....	94
2.7.2	包含、格式化和统计 变更 .....	52	4.2.2	文件转换（内容过滤） .....	97
2.7.3	贡献统计 .....	54	4.2.3	关键字替换表达式 .....	99
2.7.4	查看文件修订 .....	55	4.2.4	其他内置属性 .....	101
2.8	小结 .....	56	4.2.5	属性宏定义 .....	101
第 3 章	使用 Git 进行程序开发 .....	58	4.3	使用 reset 命令修复错误 .....	102
3.1	新建提交 .....	58	4.3.1	回退分支 head .....	102
3.1.1	新建提交的 DAG 视图 .....	59	4.3.2	重置分支 head 和索引 .....	103
3.1.2	索引——提交的暂存区 .....	60	4.3.3	丢弃变更和回退分支 .....	105
3.1.3	查看已提交的变更 .....	61	4.3.4	安全模式重置——保留 用户变更 .....	106
3.1.4	可查询的提交 .....	71	4.4	隐藏暂存变更 .....	108
3.1.5	修改提交 .....	73	4.4.1	使用 git stash .....	108
3.2	使用分支 .....	75	4.4.2	隐藏和暂存区 .....	109
3.2.1	新建分支 .....	76	4.4.3	暂存探幽 .....	110
			4.5	管理工作区和暂存区 .....	112
			4.5.1	查看文件和目录 .....	113

4.5.2 搜索文件内容 .....	114	5.4.1 推送变更到公共 版本库 .....	148
4.5.3 撤销对文件的跟踪、暂存 和修改 .....	115	5.4.2 生成 pull 请求 .....	149
4.5.4 文件版本回退 .....	116	5.4.3 交换补丁 .....	149
4.5.5 清理工作区 .....	117	5.5 信任链 .....	151
4.6 多工作目录 .....	118	5.5.1 内容地址存储 .....	152
4.7 小结 .....	119	5.5.2 轻量级标签、附注标签和 签名标签 .....	152
<b>第 5 章 Git 协作开发</b> .....	<b>120</b>	5.5.3 签名提交 .....	154
5.1 协作工作流 .....	120	5.5.4 合并签名标签 (合并标签) .....	155
5.1.1 空版本库 .....	121	5.6 小结 .....	157
5.1.2 和其他版本库交互 .....	122	<b>第 6 章 分支应用进阶</b> .....	<b>158</b>
5.1.3 中心式工作流 .....	122	6.1 分支的类型和用途 .....	158
5.1.4 对等网络或者分支 工作流 .....	123	6.1.1 长期或者永久性分支 .....	159
5.1.5 维护者和集成管理 工作流 .....	124	6.1.2 短期分支 .....	164
5.1.6 层级式(主从式) 工作流 .....	125	6.2 分支工作流和发布工程 .....	165
5.2 远程版本库管理 .....	126	6.2.1 预览或者主干分支 工作流 .....	165
5.2.1 原生的远程版本库 .....	127	6.2.2 节点或者渐进稳定性 分支工作流 .....	166
5.2.2 浏览远程版本库 .....	127	6.2.3 主题分支工作流 .....	168
5.2.3 新建远程版本库 .....	128	6.2.4 Git 流——一种成功的 Git 分支模型 .....	172
5.2.4 远程版本库信息更新 .....	129	6.2.5 修复安全问题 .....	173
5.2.5 兼容不规则工作流 .....	131	6.3 远程版本库上分支间的 交互 .....	175
5.3 传输协议 .....	132	6.3.1 上游和下游 .....	175
5.3.1 本地传输 .....	132	6.3.2 远程跟踪分支和 refspec .....	176
5.3.2 智能传输 .....	134	6.3.3 fetch、pull 和 push .....	177
5.3.3 使用 bundle 进行离线 传输 .....	136	6.3.4 拉取、推送分支和标签 .....	179
5.3.4 远程版本库传输助手 .....	142		
5.3.5 凭据/密码管理 .....	145		
5.4 发布变更到上游 .....	148		

6.3.5 推送模式应用 .....	181	8.3.2 使用笔记存储附加 信息 .....	242
6.4 小结 .....	185	8.3.3 置换机制应用 .....	249
<b>第 7 章 集成变更</b> .....	186	8.4 小结 .....	253
7.1 集成变更的方法 .....	186	<b>第 9 章 子项目管理——构建活动 框架</b> .....	254
7.1.1 合并分支 .....	187	9.1 管理库和框架的依赖 .....	255
7.1.2 拷贝和应用变更集 .....	191	9.1.1 Git 外部依赖管理 .....	256
7.1.3 分支变基 .....	194	9.1.2 手工导入项目代码 .....	257
7.2 解决合并冲突 .....	197	9.1.3 包含子项目代码的 Git 子树 .....	258
7.2.1 三路合并 .....	198	9.1.4 子模块解决方案—— 版本库嵌套 .....	267
7.2.2 检测失败的合并操作 .....	199	9.1.5 将子文件夹迁移到子树 或者子模块中 .....	279
7.2.3 避免合并冲突 .....	203	9.1.6 子树和子模块 .....	280
7.2.4 处理合并冲突 .....	205	9.2 大型 Git 版本库管理 .....	283
7.3 小结 .....	207	9.2.1 处理包含大量历史记录的 版本库 .....	283
<b>第 8 章 历史记录管理</b> .....	209	9.2.2 处理包含大量二进制文件 的版本库 .....	285
8.1 Git 内部机制简介 .....	210	9.3 小结 .....	287
8.1.1 Git 对象 .....	210	<b>第 10 章 Git 的定制和扩展</b> .....	288
8.1.2 Git 的底层命令和高层 命令 .....	213	10.1 Git 与命令行 .....	289
8.1.3 Git 环境变量 .....	213	10.1.1 Git 命令行提示符 .....	289
8.2 重写修订历史 .....	216	10.1.2 Git 命令自动补全 .....	292
8.2.1 编辑最后一次提交 .....	217	10.1.3 Git 命令自动校正 .....	293
8.2.2 交互式变基 .....	218	10.1.4 命令行美化 .....	294
8.2.3 外部工具——补丁管理 接口 .....	224	10.1.5 命令行工具替代 方案 .....	294
8.2.4 使用 git filter-branch 进行 脚本化重写 .....	225	10.2 图形化接口 .....	295
8.2.5 用于重写大型项目历史 记录的外部工具 .....	231	10.2.1 图形化工具种类 .....	295
8.2.6 重写已发布历史的 风险 .....	232		
8.3 历史记录的非重写式编辑 .....	236		
8.3.1 还原提交 .....	236		



10.2.2 图形化的 diff 和 merge 工具 .....	296	11.3.4 Git 版本库服务 .....	335
10.2.3 图形化接口示例 .....	298	11.3.5 Git 版本库管理工具 .....	339
10.3 配置 Git .....	299	11.3.6 版本库托管应用技巧 .....	340
10.3.1 命令行选项和环境变量 .....	299	11.4 改进开发工作流 .....	342
10.3.2 Git 配置文件 .....	299	11.5 小结 .....	342
10.3.3 使用 gitattribute 配置单个文件 .....	309	第 12 章 Git 最佳实践 .....	343
10.4 Git 自动化钩子 .....	311	12.1 启动项目 .....	343
10.4.1 安装 Git 钩子 .....	312	12.1.1 将工作分配到版本库 .....	344
10.4.2 版本库模板 .....	312	12.1.2 选择协作工作流 .....	344
10.4.3 客户端钩子 .....	313	12.1.3 选择需要实行版本控制的文件 .....	344
10.4.4 服务端钩子 .....	318	12.2 推进项目 .....	345
10.5 Git 扩展 .....	319	12.2.1 使用主题分支 .....	345
10.5.1 Git 命令行别名 .....	319	12.2.2 确定工作背景 .....	346
10.5.2 添加新的 Git 命令 .....	321	12.2.3 将变更分解成独立的逻辑单元 .....	347
10.5.3 凭据助手和远程版本库助手 .....	322	12.2.4 编写简洁易读的注释 .....	347
10.6 小结 .....	322	12.2.5 为提交变更做好准备 .....	349
第 11 章 Git 日常管理 .....	323	12.3 集成变更 .....	349
11.1 版本库维护 .....	324	12.3.1 提交和描述变更 .....	349
11.2 数据恢复和故障诊断 .....	325	12.3.2 审核变更的艺术 .....	351
11.2.1 恢复已丢弃的提交记录 .....	325	12.3.3 处理审核结果和评论 .....	353
11.2.2 Git 故障诊断 .....	327	12.4 其他注意事项 .....	353
11.3 Git 服务端配置 .....	328	12.4.1 不用慌, 一切几乎都是可以恢复的 .....	354
11.3.1 服务端钩子 .....	328	12.4.2 不要修改已发布的历史记录 .....	354
11.3.2 使用钩子实现 Git 强制策略 .....	332	12.4.3 版本发布的数字化和标签化 .....	354
11.3.3 签名推送 .....	334	12.4.4 尽可能自动化 .....	355
		12.5 小结 .....	355

# 第 1 章

## Git 应用入门

本书是专门为 Git 初学者和高级用户撰写的，希望能够在他们精通 Git 要义的道路上有有所帮助。因此，接下来的章节会假定读者已经了解了 Git 的基本使用，并且度过了学习 Git 的新手阶段。

本章的内容可以作为 Git 版本控制基础知识的简单回顾。本章的重点会放在实际应用方面，通过开发一个简易示例项目，演示和解说基本的版本控制操作，以及两个开发者之间的协作流程。

本章将会介绍以下知识。

- 搭建 Git 环境和创建 Git 版本库 (init、clone)。
- 文件添加、状态检查、创建注释和查看历史记录。
- 与其他 Git 版本库交互 (pull、push)。
- 解决合并冲突。
- 创建分支列表、列表切换和合并。
- 创建标签。

### 1.1 版本控制与 Git

版本控制系统（有时也称修订控制）是一种用户可以根据时间追溯项目文件（存放于版本库中）修改历史和属性的工具，它还可以帮助团队成员协作开发。当前流行的版本控制系统可以为每个开发人员提供专属的沙箱，防止他们的工作发生冲突，同时采用冲突合并和同步机制，实现以非阻塞的方式进行高效协作。

像 Git 这类分布式版本控制系统为每个用户提供专属于其自己的项目历史副本、版本库的副本。Git 系统如此高效的原因有以下几个：首先，几乎所有操作都是在用户本机上执行，而且非常灵活；其次，你可以使用多种方式建立版本仓库。版本仓库对于开发来说意味着每个开发人员都有整个项目文件的独立工作区（也称工作目录）。Git 采用的分支模型支持本地分支，而且分支的发布也非常灵活，用户可以使用分支进行内容切换，还可以在开发过程中将不同工作放置于相互隔离的沙箱中（有可能构建出独立、灵活的主题分支工作流）。

事实上，版本库的整个变更历史都是可以访问的，用户可以撤销或者回退更改过的内容到最后一个工作版本等。当每个修改被提交之后，用户的修改提交记录也被记录下来，因此提交代码修改的用户也就很容易被定位。你还可以比较文件的不同版本，将代码回退到某个用户提交 bug 报告之前的版本，甚至可以知道哪个版本的变更导致了上述 bug。其实，Git 主要是通过 **reflog** 命令来跟踪分支的变更记录信息并实现回退和覆盖目的的。

Git 有一个独特的功能是它支持显式访问暂存区以便创建注释（对项目进行新的修订）。这为用户管理工作区和确定将来的注释信息带来了更多灵活性。

所有这些灵活、强大的特性都是要付出代价的。虽然掌握 Git 的基本使用非常简单，但是精通 Git 的使用并不是那么容易。本书将会帮助你在成为 Git 专家的道路上披荆斩棘，不过在此让我们先回顾一下 Git 的基本使用。

## 1.2 Git 简易示例

让我们通过两个开发人员在简单项目上使用 Git 进行协作开发来一步一步地构建一个简单示例。读者可以从 <http://www.packtpub.com> 下载相关的项目示例代码。你会发现本章的示例代码文件中包含 3 个版本库（一个是服务端的，另外两个是开发者的），而且你可以浏览版本库的代码、查询修改历史、执行 **reflog** 命令等。

### 1.2.1 创建版本库

某公司准备研发一款新产品。该产品主要的用途是从特定区间内随机获取若干个整数。

该公司指派了两名开发人员负责这个新项目，他们的名字分别是 Alice 和 Bob。两名远程办公的开发人员经过和公司领导协商之后，决定使用 C 语言开发一套命令行应用来完成该产品的研发，并且使用 Git 2.5.0 (<http://git-scm.com/>) 进行程序代码版本控制管理。该产品的用途主要是用来进行过程演示的，而且也非常简单。程序代码的细节并不是重点，我

们关注的是如何管理代码变更。

团队规模很小，因此他们决定以图 1-1 所示的组织结构启动项目。

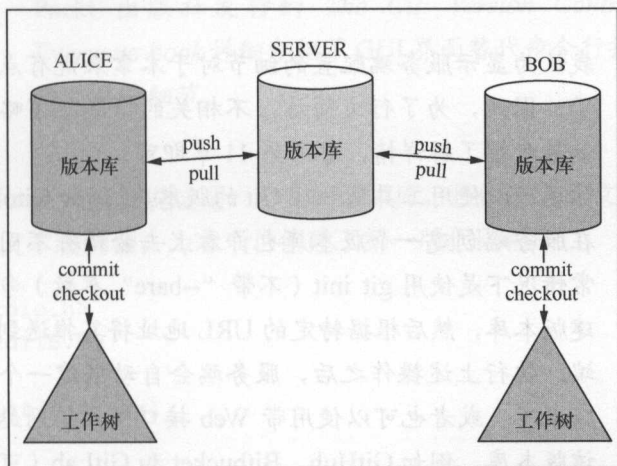


图 1-1 启动项目的组织结构



这个启动配置非常自由，虽然有中心版本库，但是却没有人负责维护它（项目启动之初，所有开发者的角色都是平等的）。启动项目的组织形式多样，如果你希望深入了解与此有关的知识，可以参考第 5 章的内容。

## 1.2.2 创建 Git 版本库

Alice 在项目启动时请求管理员 Carol 为她创建一个新的版本库以方便团队协作。



命令行示例遵循的是 UNIX 系统风格，命令行前面的提示信息由“用户名@主机名/文件目录”组成，这样一眼就可以看出由谁执行该命令，属于哪台计算机以及文件目录是什么。上述风格在 UNIX 环境中很常见（在 Linux 系统下也是如此）。可以参考第 10 章的内容，让 Git 系统显示特定信息，例如版本库名称、版本库下的子目录名、当前分支，甚至工作区的状态信息。



```
carol@server ~$ mkdir -p /srv/git
carol@server ~$ cd /srv/git
carol@server /srv/git$ git init --bare random.git
```



我认为显示服务端配置的细节对于本章来说有点多余了。因此，为了行文简洁，不相关的信息就省略了。如果希望了解详情，参考第 11 章即可。

你还可以使用工具来管理 Git 的版本库(例如 Gitolite)，在服务端创建一个版本库也许看上去会稍有不同。通常情况下是使用 `git init` (不带 “--bare” 参数) 命令创建版本库，然后根据特定的 URL 地址将其推送到服务端，执行上述操作之后，服务端会自动创建一个公共版本库。或者也可以使用带 Web 接口工具的网站创建该版本库，例如 GitHub、Bitbucket 和 GitLab (可以托管或内部部署)。

## 1.2.3 克隆版本库并添加注释

Bob 知道项目版本库就绪的消息之后，就开始了编写代码的工作。

因为这是他在本项目中首次使用 Git，因此，他在自己的版本库根目录下建立了对应的 `~/.gitconfig` 文件，该文件主要是用来帮助他标记日志文件中特定的注释信息：

```
[user]
  name = Bob Hacker
  email = bob@company.com
```

现在他需要获取自己的版本库实例：

```
bob@hostB ~$ git clone https://git.company.com/random
Cloning into random...
Warning: You appear to have cloned an empty repository.
done.
bob@hostB ~$ cd random
bob@hostB random$
```



本章所有示例使用的都是命令行接口。这些命令也可以使用 Git 的 GUI 应用程序或者 IDE 集成环境完成。Packt 出版社发行的 *The Git: Version Control for Everyone Book* 详细介绍了 GUI 界面替代命令行执行任务的具体细节。

Bob 注意到 Git 系统提示说这是一个空的版本库，还没有代码文件，所以他开始编写代码了。他打开了文本编辑器，为本项目创建了第一版代码程序：

```
#include <stdio.h>
#include <stdlib.h>

int random_int(int max)
{
    return rand() % max;
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int max = atoi(argv[1]);

    int result = random_int(max);
    printf("%d\n", result);

    return EXIT_SUCCESS;
}
```

一般来说，和大部分原始程序实现类似，该程序还非常简陋。不过这是一个很好的开端。在提交代码之前，Bob 希望这个程序可以通过编译并能够运行：

```
bob@hostB random$ gcc -std=c99 random.c
bob@hostB random$ ls -l
total 43
-rwxr-xr-x 1 bob staff 86139 May 29 17:36 a.out
-rw-r--r-- 1 bob staff 331 May 19 17:11 random.c
```

```

bob@hostB random$ ./a.out
Usage: ./a.out <number>
bob@hostB random$ ./a.out 10
1

```

好的！现在把该文件添加到版本库中：

```
bob@hostB random$ git add random.c
```

Bob 使用 “status” 命令来确保先前的所有修改都没什么问题：



为了节省示例篇幅，我们截取了 git status 命令的部分内容。你可以在本章后续内容中看到执行该命令后的完整内容输出。

```

bob@hostB random$ git status -s
A random.c
?? a.out

```

Git 系统显示了警告信息，因为它不知道该如何处理 a.out 文件：它既没有在跟踪列表中，也没有在忽略列表中。它是编译过程中生成的可执行体，不应该存放于版本库中。Bob 可以暂时不用理会这个提示信息。

现在，向服务端提交（commit）该代码文件：

```

bob@hostB random$ git commit -a -m "Initial implementation"
[master (root-commit) 2b953b4] Initial implementation
1 file changed, 22 insertions(+)
Create mode 100644 random.c

```



一般来说，在添加注释信息时，不仅可以使 -m <信息> 命令行选项，还可以让 Git 打开一个文本编辑器完成相关操作。本示例使用这种格式是为了让示例代码看起来更紧凑。

-a/--all 选项的意思是接受被追踪文件的所有变更，你可以在暂存区创建一个注释来实现操作隔离，不过这是另外一个问题了。详情可以参考第 4 章内容。

## 1.2.4 发布修改

在完成项目的初始版本之后，Bob 准备发布它们（提交到服务器，供团队其他成员访问）。他将自己的工作成果推送到了服务端：

```
bob@hostB random$ git push
warning: push.default is unset; its implicit value has changed in
Git 2.0 from 'matching' to 'simple'. To squelch this message [...]
To https://git.company.com/random
* [new branch]      master -> master
bob@hostB random$ git config --global push.default simple
```

### 注意：



Git 会根据用户的网络速度，显示诸如 clone、push 和 fetch 操作的具体进度。为了简单起见，本书示例省略了这些内容，当需要查看代码文件历史和变更记录时才会显示相关信息。

## 1.2.5 查看历史版本

因为这是 Alice 第一次在她的个人电脑上使用 Git，因此，她必须告诉 Git 系统如何识别她提交的注释：

```
alice@hostA ~$ git config --global user.name "Alice Developer"
alice@hostA ~$ git config --global user.email alice@company.com
```

现在 Alice 需要建立专属于她自己的版本库实例：

```
alice@hostA ~$ git clone https://git.company.com/random
Cloning into random...
done.
```

Alice 打算查看一下工作目录：

```
alice@hostA ~$ cd random
alice@hostA random$ ls -al
total 1
drwxr-xr-x  1 alice staff  0 May 30 16:44 .
drwxr-xr-x  4 alice staff  0 May 30 16:39 ..
drwxr-xr-x  1 alice staff  0 May 30 16:39 .git
```



```
-rw-r--r--      1 alice staff    353 May 30 16:39 random.c
```



.git 目录下包含 Alice 的版本库的拷贝（克隆），并且这些文件是以 Git 内部格式存在的，同时还包含一些针对版本库的管理信息。你可以在 Git 帮助手册中的 gitrepository-layout (5) 章节找到文件格式的详细说明，只需要在命令行中键入 `git help repository-layout` 命令即可。

她希望查看日志的细节信息（查看项目历史记录）：

```
alice@hostA random$ git log
commit 2b953b4e80abfb77bdcd94e74dedeebf6aba870
Author: Bob Hacker <bob@company.com>
Date:   Thu May 29 19:53:54 2015 +0200
```

Initial implementation



#### 修订追踪：

在最底层实现中，Git 历史版本识别是通过一个 SHA-1 哈希码实现的，例如 2b953b4e80。Git 支持多种形式的版本查询，其中就包括 SHA-1 码精确匹配（最少提供 4 个字符）。请参考第 2 章了解详情。

当 Alice 决定浏览一遍代码时，她突然发现了一个严重的问题：随机数生成部分一直都没有初始化！她经过一个快速测试发现程序生成的结果都是同一个数。幸运的是，她并不需要修改 `main()` 内部的代码，只需要在顶部加入相应的 `#include` 引用即可：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
int random_int(int max)
{
    return rand() % max;
}
```

```

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <number>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int max = atoi(argv[1]);

    srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);

    return EXIT_SUCCESS;
}

```

改完代码之后，她运行了几次程序，来确认程序真的可以生成随机数。一切似乎都进行得很顺利。她使用 `git status` 命令查看了之前的文件变更：

```

alice@hostA random$ git status -s
M random.c

```

不必感到大惊小怪。Git 知道 `random.c` 文件被修改了。然后 Alice 使用 `git diff` 命令再次确认对代码的修改：



从现在开始，我们将不会显示未被跟踪的文件信息，除非它和讨论的主题相关。现在假定 Alice 已经生成了一个忽略配置文件。该配置文件的详情，可以参考第 4 章。

```

alice@hostA random$ git diff
diff --git a/random.c b/random.c
index cc09a47..5e095ce 100644
--- a/random.c
+++ b/random.c
@@ -1,5 +1,6 @@
#include <stdio.h>
#include <stdlib.h>
+#include <time.h>

```

```

int random_int(int max)
{
@@ -15,6 +16,7 @@ int main(int argc, char *argv[])

    int max = atoi(argv[1]);

+   srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);

```

现在，可以提交变更，然后将它们发布到公共版本库了：

```

alice@hostA random$ git commit -a -m "Initialize random number generator"
[master db23d0e] Initialize random number generator
1 file changed, 2 insertions(+)
alice@hostA random$ git push
To https://git.company.com/random
3b16f17..db23d0e master -> masterRenaming and moving files

```

## 1.2.6 重命名、移动文件

Bob 接下来的工作是重构工作区目录。他不希望版本库的顶层目录文件太多，所以他决定将所有源代码文件移动到“src/”子目录下：

```

bob@hostA random$ mkdir src
bob@hostA random$ git mv random.c src/
bob@hostA random$ git status -s
R   random.c -> src/random.c
bob@hostA random$ git commit -a -m "Directory structure"
[master 69e0d3d] Directory structure
1 file changed, 0 insertions(+), 0 deletions(-)
rename random.c => src/random.c (100%)

```

现在，他为了确保目录重构之后使用“diff”命令输出结果的差异不至于太大，将 Git 系统配置为始终执行重命名和拷贝的检测：

```

bob@hostB random$ git config --global diff.renames copies

```

Bob 觉得是时候为项目添加一个合适的 Makefile 配置文件，以及一个 README 帮助文件了：

```

bob@hostA random$ git add README Makefile

```

```

bob@hostA random$ git status -s
A   Makefile
A   README
bob@hostA random$ git commit -a -m "Added Makefile and README"
[master abfee4] Added Makefile and README
2 files changed, 15 insertions(+)
create mode 100644 Makefile
create mode 100644 README

```

Bob 将 “random.c” 文件的名字改为 “rand.c”:

```

bob@hostA random$ git mv src/random.c src/rand.c

```

上述操作当然也需要修改 Makefile 文件:

```

bob@hostA random$ git status -s
M   Makefile
R   src/random.c -> src/rand.c

```

然后, 他提交了这些修改。

## 1.2.7 更新版本库 (合并)

项目文件重组完成之后, Bob 打算将这些变更推送到服务端:

```

bob@hostA random$ git push
$ git push
To https://git.company.com/random
! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'https://git.company.com/random'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
hint: pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
details.

```

但是 Alice 同时也在该项目中工作, 而且她已经向服务端推送了自己的代码。Git 系统现在不允许 Bob 推送他对项目代码的变更, 因为 Alice 已经推送了一些内容到 master 分支上, 系统会保护她提交的变更。





为了行文简洁，Git 命令行中输出的提示和帮助信息在后续篇章中会略去。

Bob 使用 “pull” 命令将服务端版本库的内容与自己的版本库同步（像命令行提示信息建议的那样）：

```
bob@hostB random $ git pull
From https://git.company.com/random
+ 3b16f17...db23d0e master -> origin/master
Auto-merging src/rand.c
Merge made by the 'recursive' strategy.
src/rand.c | 2 ++
1 file changed, 2 insertions(+)
```

执行 “pull” 命令之后，Git 系统会将服务端版本库中的变更下载到 Bob 本机，然后自动将它们和 Bob 本机版本库的变更合并，最后把合并后的变更提交到本机版本库中。

现在万事俱备了：

```
bob@hostB random$ git show
commit ba5807e44d75285244e1d2each1c10cbc5cf3935
Merge: 3b16f17 db23d0e
Author: Bob Hacker <bob@company.com>
Date: Sat May 31 20:43:42 2015 +0200
```

```
Merge branch 'master' of https://git.company.com/random
```

合并后的提交也完成了。Git 系统可以直接将 Alice 提交的变更与 Bob 移动或重命名后的文件合并，是不是很神奇呢？

Bob 检查编译（因为自动合并之后并不能绝对保证代码没问题）了一下变更合并后的代码，准备将合并变更后的代码推送到服务端：

```
bob@hostB random$ git push
To https://git.company.com/random
db23d0e..ba5807e master -> master
```

## 1.2.8 创建标签

Alice 和 Bob 认为项目可以进行更大范围的发布了。Bob 创建了一个标签（tag），以便

日后他们方便地访问/引用发布过的预览版本。他为此使用了一个带注释的标签，当然大家一般采用的替代性方案是使用带数字签名的标签，该标签通常会包含一个PGP数字签名（以后的验证需要用到它）：

```
bob@hostB random$ git tag -a -m "random v0.1" v0.1
bob@hostB random$ git tag --list
v0.1
bob@hostB random$ git log -1 --decorate --abbrev-commit
commit ba5807e (HEAD -> master, tag: v0.1, origin/master)
Merge: 3b16f17 db23d0e
Author: Bob Hacker <bob@company.com>
Date: Sat May 31 20:43:42 2015 +0200
```

Merge branch 'master' of https://git.company.com/random

当然，v0.1版的标签如果只放在Bob本地的版本库中是没有什么意义的。接下来他将刚创建的标签推送到服务端：

```
bob@hostB random$ git push origin tag v0.1
Counting objects: 1, done.
Writing objects: 100% (1/1), 162 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (1/1), done.
To https://git.company.com/random
* [new tag] v0.1 -> v0.1
```

Alice 为了获得这个标签，更新了她的版本库，然后开始了她的日常工作：

```
alice@hostA random$ git pull
From https://git.company.com/random
f4d9753..be08dee master -> origin/master
* [new tag] v0.1 -> v0.1
Updating f4d9753..be08dee
Fast-forward
 Makefile | 11 ++++++++
 README | 4 +++
 random.c => src/rand.c | 0
3 files changed, 15 insertions(+)
create mode 100644 Makefile
create mode 100644 README
rename random.c => src/rand.c (100%)
```

## 1.2.9 解决合并冲突

Alice 认为将生成伪随机数功能修改成一个单独的子程序是个好主意。这样一来，初始化和生成随机数的功能都独立封装起来了，将来需求变更的时候更容易实现一些。她决定给程序添加一个 `init_rand()` 函数：

```
void init_rand(void)
{
    srand(time(NULL));
}
```

接下来，编译运行一下代码，看看有没有什么问题：

```
alice@hostA random$ make
gcc -std=c99 -Wall -Wextra -o rand src/rand.c
alice@hostA random$ ls -F
Makefile rand* README src/
```

通过编译，程序没什么问题之后就可以提交变更了：

```
alice@hostA random$ git status -s
M src/rand.c
alice@hostA random$ git commit -a -m "Abstract RNG initialization"
[master 26f8e35] Abstract RNG initialization
1 files changed, 6 insertions(+), 1 deletion(-)
```

从提示信息可以看到，提交成功了。

同时，Bob 在与 `rand()` 函数相关的开发文档中发现，它是用来生成简单的伪随机数的标准函数，可能并不能满足实际需要：

```
bob@hostB random$ git pull
Already up-to-date.
```

他决定在提交代码变更的注释中添加对此问题的备注说明：

```
bob@hostB random$ git status -s
M src/rand.c
bob@hostB random$ git diff
diff --git a/src/rand.c b/src/rand.c
index 5e095ce..8fddf5d 100644
```

```

--- a/src/rand.c
+++ b/src/rand.c
@@ -2,6 +2,7 @@
#include <stdlib.h>
#include <time.h>

+// TODO: use a better random generator
int random_int(int max)
{
    return rand() % max;
}

```

他提交代码的变更之后，将它们推送到了服务端：

```

bob@hostB random$ git commit -m 'Add TODO comment for random_int()'
[master 8c4ceca] Use Add TODO comment for random_int()
1 files changed, 1 insertion(+)
bob@hostB random$ git push
To https://git.company.com/random
ba5807e..8c4ceca master -> master

```

因此，当 Alice 准备推送她的变更到服务端时，Git 系统拒绝了该操作：

```

alice@hostA random$ git push
To https://git.company.com/random
! [rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'https://git.company.com/random'
[...]

```

Bob 一定是推送了不少变更到服务端。Alice 需要再次从服务端的版本库上下载最新版本的项目文件，然后亲自把自己的变更和 Bob 的变更合并：

```

alice@hostA random$ git pull
From https://git.company.com/random
ba5807e..8c4ceca master -> origin/master
Auto-merging src/rand.c
CONFLICT (content): Merge conflict in src/rand.c
Automatic merge failed; fix conflicts and then commit the result.

```

该合并操作并没有像上次那样被顺利执行。Git 系统无法自动合并 Alice 和 Bob 两人提交的变更。显然，文件变更之间有冲突。Alice 决定用文本编辑器打开文件“src/rand.c”一探究竟（她也可以使用图形化的合并工具查看代码差异）：



```

<<<<<<< HEAD
void init_rand(void)
{
    srand(time(NULL));
}

=====
// TODO: use a better random generator
>>>>>>> 8c4ceca59d7402fb24a672c624b7ad816cf04e08
int random_int(int max)

```

Git 系统中现在既有 Alice 提交的代码变更（在<<<<<<< HEAD 和===== 冲突标记之间），也有 Bob 提交的代码变更（在=====和 >>>>>>>之间）。我们希望的结果是将两人的代码融为一体。Git 系统无法自动合并它们，因为这些代码块并不是独立的。Alice 的 init\_rand()函数可以简单地插入 Bob 添加的代码注释之前。执行上述操作之后，结果如下：

```

alice@hostA random$ git diff
diff --cc src/rand.c
index 17ad8ea,8fddf5d..0000000
--- a/src/rand.c
+++ b/src/rand.c
@@@ -2,11 -2,7 +2,12 @@@
    #include <stdlib.h>
    #include <time.h>

+void init_rand(void)
+{
+    srand(time(NULL));
+}
+

+ // TODO: use a better random generator
+ int random_int(int max)
+ {
+     return rand() % max;
+ }

```

这样冲突应该就可以解决了。Alice 重新编译运行了一下程序，然后提交了这一变更：

```

alice@hostA random$ git status -s
UU src/rand.c
alice@hostA random$ git commit -a -m 'Merge: init_rand() + TODO'

```

```
[master 493e222] Merge: init_rand() + TODO
```

然后她尝试将该变更推送到服务端版本库:

```
alice@hostA random$ git push
To https://git.company.com/random
8c4ceca..493e222 master -> master
```

大功告成!

## 1.2.10 添加和移除文件

Bob 打算给项目添加一个和版权声明有关的 COPYRIGHT 文件, 当然还打算添加一个记录软件新特性的文件 (不过还没有创建), 所以他使用批处理命令添加了工作区中的所有文件到版本库中:

```
bob@hostB random$ git add -v
add 'COPYRIGHT'
add 'COPYRIGHT~'
```

因为 Bob 没有配置的忽略模式, 因此作为备份文件的 “COPYRIGHT~” 也被提交到了版本库。接下来我们移除该文件:

```
bob@hostB random$ git status -s
A  COPYRIGHT
A  COPYRIGHT~
bob@hostB random$ git rm COPYRIGHT~
error: 'COPYRIGHT~' has changes staged in the index
(use --cached to keep the file, or -f to force removal)
bob@hostB random$ git rm -f COPYRIGHT~
rm 'COPYRIGHT~'
```

检查一下文件状态, 然后提交变更:

```
bob@hostB random$ git status -s
A  COPYRIGHT
bob@hostB random$ git commit -a -m 'Added COPYRIGHT'
[master ca3cdd6] Added COPYRIGHT
1 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 COPYRIGHT
```

## 1.2.11 撤销对单个文件的修改

百无聊赖之际，Bob 决定调整一下文件 `rand.c` 的代码缩进格式，使之符合统一的命名规范。

```
bob@hostB random$ indent src/rand.c
```

他统计了一下该文件源代码的变更记录：

```
bob@hostB random$ git diff --stat
src/rand.c | 40 ++++++-----
1 files changed, 22 insertions(+), 18 deletions(-)
```

样式调整的变更太多了（对于如此小的文件来说），在合并时可能会出问题。Bob 冷静了一下，然后撤销了对文件 `rand.c` 的样式调整：

```
bob@hostB random$ git status -s
M src/rand.c
bob@hostB random$ git checkout -- src/rand.c
bob@hostB random$ git status -s
```



如果你不记得如何回退一个特定类型的变更或者更新某个已提交的变更（使用不带“-a”参数的命令“`git commit`”），执行“`git status`”命令（不带“-s”参数）后的输出结果会包含如下所示的帮助信息：

```
bob@hostB random$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    modified:   src/rand.c
```

## 1.2.12 创建新分支

Alice 注意到代码中取模运算返回给定区间内随机数的分布并不是均匀的, 因为大部分情况下返回的都是较小的数。她打算修复这个问题。为了将相关的开发工作和代码中的其他变更隔离, 她决定在自己的版本库中创建一个新分支(分支的具体使用可以参考第 6 章), 然后切换到该分支:

```
alice@hostA random$ git checkout -b better-random
Switched to a new branch 'better-random'
alice@hostA random$ git branch
* better-random
  master
```



除了使用 “git checkout -b better-random” 命令创建一个新分支, 然后使用命令切换到该分支之外, 她还可以首先使用 “git branch better-random” 命令创建一个新分支, 然后使用 “git checkout better-random” 命令切换到该分支。

她决定使用 `RAND_MAX` 常量控制 `rand()` 函数生成随机数的范围。相关的代码修改如下:

```
alice@hostA random$ git diff
diff --git a/src/rand.c b/src/rand.c
index 2125b0d..5ded9bb 100644
--- a/src/rand.c
+++ b/src/rand.c
@@ -10,7 +10,7 @@ void init_rand(void)
 // TODO: use a better random generator
 int random_int(int max)
 {
-    return rand() % max;
+    return rand()*max / RAND_MAX;
 }

 int main(int argc, char *argv[])
```

她提交了上述代码变更, 然后将它们推送到了服务端, 她知道上述推送操作可以顺利



执行，因为这些操作都是在她的私有分支上进行的：

```
alice@hostA random$ git commit -a -m 'random_int: use rescaling'
[better-random bb71a80] random_int: use rescaling
1 files changed, 1 insertion(+), 1 deletion(-)
alice@hostA random$ git push
fatal: The current branch better-random has no upstream branch.
To push the current branch and set the remote as upstream, use

git push --set-upstream origin better-random
```

通过上述信息可以知道，Git 系统希望 Alice 为她新创建的分支（它采用的推送策略是 simple 模式）在远程版本库中添加对应的上游分支，这样可以让分支推送到远程分支的目标更明确。

```
alice@hostA random$ git push --set-upstream origin better-random
To https://git.company.com/random
* [new branch] better-random -> better-random
```



如果她希望更直观地管理她自己的分支结构并且只对自己可见，那么她需要配置好与服务端的相关映射，或者使用诸如 Gitolite 之类 Git 版本库管理软件来管理自己的分支。

### 1.2.13 合并分支（无冲突）

与此同时，在默认的主分支下，Bob 打算推送自己给项目添加 COPYRIGHT 文件的变更：

```
bob@hostB random$ git push
To https://git.company.com/random
! [rejected] master -> master (non-fast-forward)
[...]
```

出现上述错误提示是因为 Alice 当时正在忙着将初始化生成随机数的部分代码封装为一个子程序（解决合并冲突），她首先向服务端版本库推送了自己的变更：

```
bob@hostB random$ git pull
From https://git.company.com/random
8c4ceca..493e222 master -> origin/master
```

```
* [new branch]      better-random -> origin/better-random
Merge made by 'recursive' strategy.
src/rand.c | 7 ++++++-
1 file changed, 6 insertions(+), 1 deletion(-)
```

Git 系统可以轻松地将 Alice 推送的变更合并，但是现在出现了一个新分支。接下来我们来看看具体细节。为了节省篇幅，下文只显示和该新分支“better- random”相关的内容（双点符号的详细用法可以参考第 2 章）：

```
bob@hostB random$ git log HEAD..origin/better-random
commit bb71a804f9686c4bada861b3fcd3cfb5600d2a47
Author: Alice Developer <alice@company.com>
Date:   Sun Jun 1 03:02:09 2015 +0200
```

```
random_int: use rescaling
```

有趣的是，Bob 希望从服务端获取 Alice 创建的新分支然后合并到自己的版本库的默认分支下（该新分支在远程版本库上已经存在）：

```
bob@hostB random$ git merge origin/better-random
Merge made by the 'recursive' strategy.
src/rand.c | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

## 1.2.14 撤销未发布的合并

Bob 意识到何时将该特性添加到主分支中应该由 Alice 来做决定。他打算撤销上述合并操作。因为这些变更还没有发布，因此只需要简单地将自己的本地主分支回退到上一次提交的变更状态即可：

```
bob@hostB random$ $ git reset --hard @{1}
HEAD is now at 3915cef Merge branch 'master' of https://git.company.com/
random
```



本示例演示了使用日志引用（reflog）机制实现变更回退操作。另外一种解决方案是使用“HEAD^”代替“@{1}”，这样也可以实现同样的效果。

## 1.3 小结

本章的主要内容是演示了一个小型开发团队在一个简单项目中协作开发的大致流程。

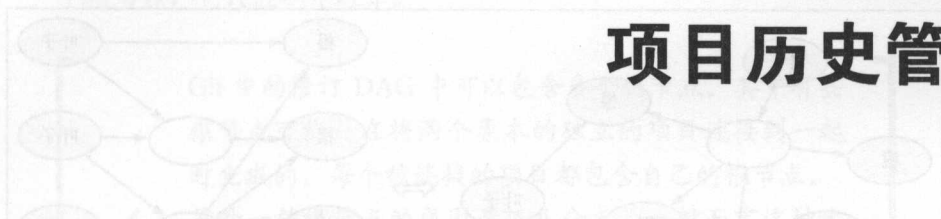
回顾了如何将 Git 和实际的研发工作集成以提高工作效率，以及创建新的版本库或者克隆一个已有的版本库；还学习了提交项目变更之前对文件的添加、编辑、移动和重命名等操作，以及如何查看项目状态和已提交的变更历史，创建预览版项目标签。

回顾了如何使用 Git 在同一项目中的团队协作，如何发布自己的变更，如何获取团队其他成员的成果。因为使用 Git 可以帮助团队成员同步开发，有时在多个团队成员之间，Git 需要用户手动解决合并冲突。

回顾了如何为软件预览版程序创建标签，以及为某一个特定功能创建独立分支。Git 需要用户显式推送标签和分支，但是获取它们是自动的。我们还学习了如何合并分支。

## 第 2 章

# 项目历史管理



项目历史记录管理是深入掌握版本控制系统最重要的组成部分之一，实际上版本控制系统最有用的一点，就是它可以以以往所有的文件版本进行归档。在这里，读者将会学到如何查询、过滤和浏览修订历史；如何引用修订（修订查询）；使用不同条件查询修订。

本章将会介绍修订中的有向无环图（Directed Acyclic Graph, DAG）概念，以及该概念和 Git 中的分支、标签和当前分支的关系。

下面是本章将要介绍的主要内容。

- 修订查询。
- 修订区间查询，历史记录约束，历史记录简化。
- 使用“锄头（pickaxe）”工具和 diff 命令查询历史记录。
- 使用 git bisect 查找 bug。
- 使用 git blame 查看文件内容历史、重命名检测。
- 查询和格式化输出结果（pretty 格式）。
- 使用短日志（shortlog）统计贡献记录。
- 使用符合 .mailmap 声明规范的作者姓名和电子邮件。
- 查看特定修订版本、diff 命令输出参数和文件修订版本。

### 2.1 有向无环图

版本控制系统有别于备份软件的显著特点是可以追踪和记录非线性的历史记录。这一



点对于团队（每个开发人员都拥有中心版本库的克隆）协作同步开发和独立并行分支的建立都是必需的。例如，某个正在修复 bug 的开发人员希望把自己对程序的变更与稳定版本的软件隔离开来，那么独立分支功能就可以完全满足需要。版本控制系统（Version Control System, VCS）可以为这种非线性的开发流程建模，而且采用了某些数据结构表征多个修订版本。

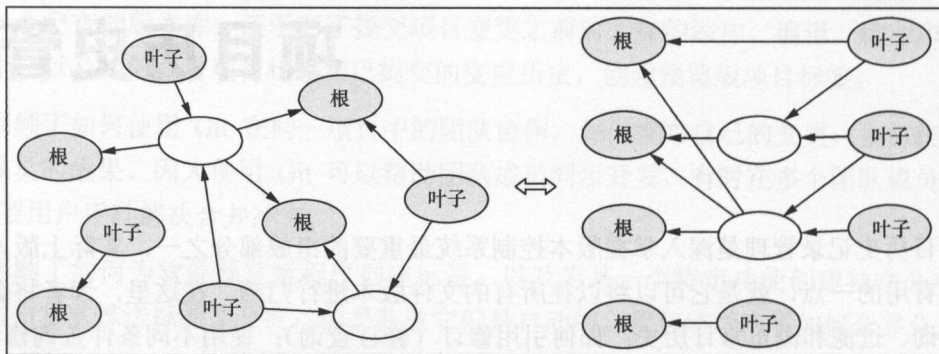


图 2-1 一个常见的有向无环图（DAG）示例。图中两边的图形结构表达的含义是一样的：

左边的形式松散，右边是按照从左到右的顺序排列的

Git 用来表示（抽象层面）项目中可能出现的非线性历史记录的结构叫有向无环图。

有向图是计算机科学（和组合数学）中节点（顶点）和有向边（箭头）一起组成的数据结构。如果一个有向图中不包含任何闭环，那么就称其为有向无环图，这意味着在有向图中无法从某个顶点出发经过若干条边返回到该点。

从图 2-1 可以看出，图中每个节点表示一个对象或者数据，每条边表示节点对象或数据之间的某种关联。

分布式版本控制系统（Distributed Version Control Systems, DVCS）中修订的 DAG 是采用如下方式表达的：

- **节点：**在 DVCS 中，每个节点代表项目的一个修订（一个版本）。这些对象通常被称为提交。
- **有向边：**在 DVCS 中，每条边是基于两个修订之间的关系生成的。箭头是从父修订指向子修订的，代表它们之间的从属关系。

有向边的表征是基于包含因果关系的两个修订而存在的。修订 DAG 中的箭头并不一定会形成闭环。通常修订的 DAG 是从左到右结构（根节点在左，叶节点在右）或者自下而上结构（最新的版本在最上面）。本书图表和 Git 官方开发文档中的 ASCII 技术示例都采

用的是从左到右的次序，而 Git 命令行中采用的则是自上而下的次序，即最新版修订在最上面。

在 DAG 中有两种非常特殊的节点类型（参见图 2-1）：

- **根节点**：这类节点没有父节点（无前驱，入度为 0）。在修订 DAG 结构中至少包含一个根节点，它代表初始版本。



Git 中的修订 DAG 中可以包含多个根节点。其中有些根节点可能是在将两个原本的独立的项目连接到一起时生成的，每个被连接的项目都包含自己的根节点。

另外一种根节点的成因是孤儿分支，一般而言这种无连接的分支都没有历史记录。例如说，GitHub 网站上通过一个版本代码库 Web 页面管理整个项目，Git 项目中会存放一组预生成的文档（帮助和 html 分支）和相关的项目（项目计划）。

- **叶节点（叶子）**：这些节点都没有孩子节点（无后继，出度为 0），而且这样的节点至少在整个结构中存在一个。它代表项目的最新版本，不包含任何新的提交。通常，修订 DAG 中的每个叶节点都有一个分支的 Head 指针指向它。

事实上，DAG 可以包含多个叶节点，这说明最新版本的对象也不是一成不变的，因为它的历史范式是线性的。

### 2.1.1 提交整个工作目录

在 DVCS 中，修订 DAG（历史记录模型）的每个节点表示项目版本的一个独立实体对象，其中包括所有文件和目录，乃至项目的整个工作区。

这意味着每个开发人员可以访问自己的版本库克隆中任意文件的历史记录，也可以选择只获取版本库的部分历史记录（浅克隆或者只克隆特定分支），也可以只签出特定文件（稀疏签出），但是无法根据日期获取版本库克隆中特定的文件历史记录。第 9 章会详细介绍如何获取版本库克隆的部分内容，例如处理大量的视频文件时，开发人员需要用到的部分只是其中很小的一个子集。

## 2.1.2 分支和标签

分支操作是开发工作在两个不同工作目录来回切换时发生的。例如，你也许希望创建一个独立分支来处理预览版程序的 bug 修复问题，将之与其他开发工作隔离开来。

标签操作是对版本库中特定修订辅以有意义的标记名称的方法。例如你也许希望为自己项目的 1.3 预览版的候选程序创建一个名为 `v1.3-rc3` 的标签。这样可以快速地回退到该版本，方便开发人员检查和验证 bug 报告等。

分支和标签有时也统称为引用（refs），它们在修订 DAG 中代表的含义是一样的。它们都是修订结构图表中的外部引用（指针），如图 2-2 所示。

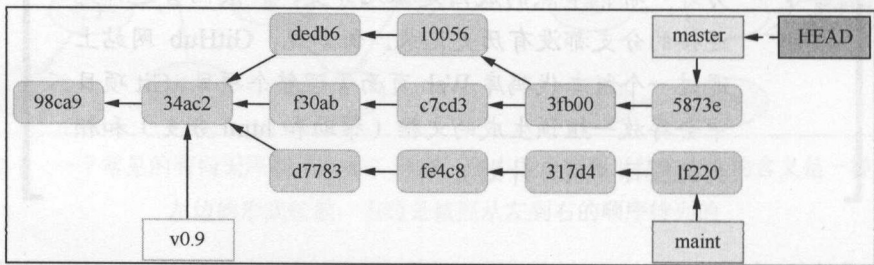


图 2-2 版本控制系统中修订的结构示意图，其中包含两个分支——`master`（当前分支）和 `maint`，其中有一个名 `v0.9` 的标签，一个包含短标识符 `34ac2` 的分支点，一个合并提交 `3fb00`

**标签**就是给定版本的符号名称（例如 `v1.3-rc3`）。它永远都指向相同对象，并且不会变更。标签的用途是，对于所有开发人员来说，都可以使用一个符号名称引用给定的修订，而且该符号对所有开发人员的意义都是一致的。查询和浏览给定的标签以获得结果，这对所有人来说都是一样的。

**分支**是一系列开发工作的符号名称。这样一系列的开发工作的最近一次提交（修订的叶节点）可以是分支的顶部或者分支的最顶端节点，甚或整个分支。创建一个新的提交将会在 DAG 中生成一个新的节点，并和相关的分支引用关联。

分支在 DAG 中代表一系列的开发工作，即所有可达的修订到修订前端（分支头部）组成的子集；换句话说，你可以沿着分支头部和与之相关的父节点遍历所有修订版本。

Git 系统在创建新的提交时当然也需要知道如何引用相关的分支节点。它需要知道当前分支是哪一个以及对应的签出工作目录。Git 采用 `HEAD` 指针达到上述目的，如图 2-2 所示。一般来说，这些指针类似访问分支的快捷方式，它们大部分情况下会指向修订 DAG 中的某些分支，间接地指向某些节点。但这也不是绝对的，你可以前往第 3 章了解 `HEAD` 指针



与分支脱离，直接指向节点的应用场景。

### 引用的全名（分支和标签）

对于引用全名的元数据来说，Git 将分支和标签存放于文件 .git 的管理区中，它们分别位于 .git/ refs/ heads/ 和 .git/ refs/ tags/ 目录下。当前的 Git 系统可以将标签和分支的信息存放于 .git/ packed- refs 文件中，以避免处理大量的小文件。不过，活动引用采用的元数据格式比较宽松——一个文件对应一个引用。

HEAD 指针（通常是一个符号引用，例如 refs/ heads/ master）一般存放于 .git/ HEAD 文件目录下。

master 分支存放于 .git/ refs/ heads/ master 目录下，而且它的全名叫 refs/ heads/ master（换句话说，分支的命名空间是 refs/ heads/ ）。分支的提示会指向分支的顶部，但是会被限制在命名空间的名称之下。对于宽松格式来说，文件内容一般是分支上当前修订的 SHA-1 标识符，即纯文本的十六进制数字。如果出现模棱两可的情况，还需要使用该标识符的全名加以区分。

类似 origin/ master 这样的远程跟踪分支，会记录远程版本库上 master 分支用户最后编辑的位置，一般会存放于 .git/ refs/ remotes/ origin/ master 文件下，并将 refs/ remotes/ origin/ master 作为其全名。远程的概念会在第 5 章详细介绍，远程跟踪分支会在第 6 章进行详细介绍。

标签 v1.3-rc3 的全名是 refs/ tags/ v1.3-rc3（标签在 refs/ tags/ 命名空间下），更精确地说是注释、附注标签，这些文件存放了指向标签对象的引用，间接指向了 DAG 中的节点，而不是直接面向提交。它是唯一一种可以指向任何对象的引用类型（二维指针）。

在命令行中可以查看这些引用的全名，例如 git show-ref。





### 2.1.3 分支点

当你为给定版本创建一个新分支时，该分支上的开发工作通常是独立的。创建新分支的行为在 DAG 图上用一个提交来表示，这使得父节点多出一个子节点，指向父节点的箭头也会相应地多出一个。



Git 在创建（拉取）分支时并不会做信息跟踪，也不会克隆或更新操作执行过程中对分支点标记保存。对应的事件信息在引用日志（根据 HEAD 创建的分支）中，不过这些信息都存放在新建分支的本地版本库中并且是临时的。此外，如果你知道 B 分支是从 A 分支衍生出来的，那么你可以使用 `git merge-base A B` 命令显示它们的分叉点；在当前的 Git 软件中还可以使用 `--fork-point` 命令选项或者使用引用日志（`reflog`）实现上述目的。

在图 2-2 中，提交 34ac2 是分支 master 和 maint 的分叉点。

### 2.1.4 合并提交

一般来说，如果你已经使用分支来进行独立的并行开发工作了，稍后又需要将分支合并。举个例子，你希望将修复了 bug 的程序变更分支集成到稳定（维护）版的主分支中（如果主分支中没有修复上述 bug）。

你也许还希望将同一项目中不同开发人员并行开发的工作成果进行统一集成，但是每个开发人员都拥有自己的版本库和一系列的成功提交。

将两种不同工作成果合并之后，会生成一个新的修订。上述操作的结果是基于多个提交的。DAG 中的一个节点表示的上述修订会出现多个父节点，这样的对象我们称为合并提交。如图 2-2 所示，节点 3fb00 就是合并提交。

## 2.2 修订内部查询

在开发过程中，你也许曾经多次试图查看项目历史中的某个修订的详情，或者希望和

当前的版本进行差异比较。查看单个修订的能力也是修订区间查询的基础，例如查看修订子集的历史记录。

很多 Git 命令都包含和修订相关的可选参数，在 Git 参考文档中它们一般是以 `rev` 为前缀的。Git 允许用户通过多种方式声明特殊的提交或者一系列的批量提交。

## 2.2.1 HEAD——最新的修订版本

大部分情况下，Git 命令需要使用参数，默认情况下使用 `HEAD`。例如说，执行 `git log` 和 `git log HEAD` 命令的输出结果是一样的。

`HEAD` 代表了当前分支，换句话说就是签出的工作目录，它也是当前工作继续进行的基础。

下面是一些和 `HEAD` 含义类似的引用，其具体含义分别如下：

- **`FETCH_HEAD`**: 记录用户最后一次执行 `git fetch` 或 `git pull` 命令拉取远程版本库的远程分支信息。它对于一次性拉取非常有用，使用一个给定 URL 拉取远程版本库的内容和使用诸如 `origin` 这样的名称拉取远程版本库内容的差别在于，我们可以使用远程跟踪分支取而代之。例如，`origin/master@{1}` 可以获取拉取版本前一版本的内容。需要注意的是，`FETCH_HEAD` 的信息会在拉取任意版本库的内容之后被重写覆盖。
- **`ORIG_HEAD`**: 记录当前分支的上一个分支节点的信息。该引用是通过命令行以极端方式（创建新提交时未设置 `ORIG_HEAD`）移动当前分支时创建的，以便能够在执行相关操作之前记录 `HEAD` 的位置。如果你希望回退或者取消这样的操作，这将是非常有用的，当然目前使用引用日志也能达到上述目的，而且用户可以在其附加信息中查询具体的使用方法。

用户还可以在特定的操作执行过程中使用临时引用：

- 在合并过程中，创建合并提交之前，`MERGE_HEAD` 会记录用户将要合并的所有分支。
- 在改写提交过程中，创建另外一个分支的改写提交之前，`CHERRY_PICK_HEAD` 会记录用户将要改写的所有提交。

## 2.2.2 分支和标签的引用

声明一个修订最简单和常见的方式是使用标识名称：分支，一系列开发工作的名称，

指向上述工作的提示指针，标签，特定修订的名称等。上述声明修订的方式可以用来查看一系列的工作历史记录，查看给定分支上最新的修订，与当前工作的分支和标签进行差异对比。

可以使用任意引用（修订 DAG 中的外部引用）查询提交记录。在 Git 命令行中可以使用分支名、标签名、远程分支作为参数查询修订历史。

通常使用分支或标签的简称就能满足需要，例如 `git log master` 命令会获取 `master` 分支的历史记录，也可以使用 `git log v1.3-rc3` 命令查找和标签为 `v1.3-rc3` 相关的历史修订记录详情。此外，有时也会出现不同引用类型名称一样的情况，例如分支和标签的名字都是 `dev`（实际开发过程中应该竭力避免这种情况发生）。有时用户在本地创建的分支名叫 `origin/master`（通常是偶然发生的），这时远程跟踪分支的简称也叫 `origin/master`，该分支一般是远程版本库的 `master` 原始版本。

在这种情况下，引用名称就会引起歧义，一般采用优先匹配的原则消除歧义，具体内容如下（下文的内容是简要版本，如希望了解详情，可以参考 `git` 帮助文档的 `git revisions`（7）章节）：

- （1）顶级的标识符名称，例如 `HEAD`。
- （2）其次是标签名称（`refs/tags/命名空间`）。
- （3）接下来是本地分支名称（`refs/heads/命名空间`）。
- （4）接下来是远程跟踪分支名称（`refs/remotes/命名空间`）。
- （5）如果远程版本库上存在默认分支名称，修订就是上述默认分支（例如可以将原生分支的 `refs/remotes/origin/HEAD` 名称作为查询参数）。

## 2.2.3 SHA-1 哈希码及其简化标识符

在 Git 中，每个修订都包含一个独一无二的标识符（对象名），即根据修订内容生成的 SHA-1 哈希码。用户可以通过 40 个字符长的十六进制数字 SHA-1 标识符查询相关的提交记录。Git 在很多地方都用到了 SHA1 标识符，例如，你可以在下列 `git log` 命令的完整日志输出记录中找到它们：

```
$ git log
commit 50f84e34a1b0bb893327043cb0c491e02ced9ff5
Author: Junio C Hamano <gitster@pobox.com>
Date: Mon Jun 9 11:39:43 2014 -0700
```



```
Update draft release notes to 2.1
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

```
commit 07768e03b5a5efc9d768d6afc6246d2ec345cace
Merge: 251cb96 eb07774
Author: Junio C Hamano <gitster@pobox.com>
Date: Mon Jun 9 11:30:12 2014 -0700
```

```
Merge branch 'jc/shortlog-ref-exclude'
```

完全给出 40 个字符长度的 SHA-1 标识符不是必需的。Git 系统很聪明，用户只需给出标识符前面的几个字符，它就能理解用户的意图，用户最少给出 4 个 SHA-1 标识符字符即可。为了使用 SHA-1 缩写标识符查询修订，给出的字符长度必须足够长从而避免产生歧义，即给定的 SHA-1 标识符的部分字符必须能够确定唯一的提交对象。

例如说 dae86e1950b1277e545cee180551750029cfe7 和 dae86e 这两个名字代表的都是同一提交对象，当然，还要假定版本库中没有其他名字是以 dae86e 开头的对象。

Git 系统在很多地方都会在其命令行输出结果中显示独一无二的 SHA-1 缩写标识符。例如前面执行 git log 命令之后的输出结果，我们可在 Merge:那一行看到 SHA-1 缩写标识符。

用户还可以通过--abbrev-提交命令选项告知 Git，使用 SHA-1 缩写标识符代替 SHA-1 标识符全名。默认情况下，Git 会使用最短为 7 个字符的 SHA-1 缩写标识符，用户可以使用上述选项参数指定标识符长度大小，例如--abbrev-commit=12。

需要注意的是，当命令行出现问题时，系统会要求使用的 SHA-1 缩写标识符尽量长一些，以便确保能够获取更精确的结果。

参数--abbrev-commit (和--abbrev 参数类似) 可以指定相关标识符的最小长度。

#### 关于 SHA-1 缩写标识符：

一般来说，在项目中使用 8 到 10 个字符长度的标识符就可以满足需要了。对于 Linux 内核这种超大型项目，才开始需要从 40 个字符长度的标识符中取 12 个字符来确保对象的唯一性。对于哈希冲突来说，两个修订版本（对象）包含相同的 SHA-1 标识符的概率是很小的，概率约为  $1/2^{80} \approx 1/(1.2 \times 10^{24})$ 。出现重复的 SHA-1 缩写标识符的可能性是和版本库规模成正比的。





SHA-1 及其缩写标识符经常会出现出现在命令行的输出结果中，用户会将之拷贝和粘贴，作为另外一个命令的参数。它们可以用于开发者之间的交流来防止产生歧义，因为 SHA-1 标识符在任何版本库的克隆中都是一样的。图 2-2 的修订 DAG 中采用了 5 个字符长度的 SHA-1 缩写标识符。

## 2.2.4 父引用

声明修订的另外一种常见方法是通过其父引用。可以通过某些子节点开始声明一个提交（例如从当前的提交 HEAD、分支头部或者标签等），然后可以根据该提交找到与之相关的父提交。有一个特殊的后缀语法来指定这样的父路径。

如果用户在修订名后面紧接着输入^符号，那么 Git 会将之视为该修订的第一个父对象。例如，HEAD^代表 HEAD 的父对象（节点），即上一个提交。

这实际上是一种快捷方式的语法。对于合并提交来说，我们拥有多个父对象，你也许希望查看其中任意一个父对象。为了查询多个父对象中的某一个，你需要在^字符后指定它的数字代号；使用^<n>意味着查看修订的第 n 个父对象。我们可以将^理解为^1 的快捷方式。

一个比较特殊的情况是，^0 指代的是该提交自身。其重要性只有在使用分支名作为参数和使用其他修订标识符产生歧义时才会体现出来。它还可以用来获取提交中包含附注（签名）的标签指针；比较 git show v0.9 和 git show v0.9^0 两个命令的输出结果差异。

这种后缀语法还可以组合使用。用户可以使用 HEAD^^来指向 HEAD 的祖父对象，即 HEAD^的父对象。

还有另外一种声明父对象的链式表达。除了输入 n 个^后缀，例如^^...^或^1^1...^1，用户还可以使用~<n>。有一个特殊情况是~和~1 是等价的，例如 HEAD~和 HEAD^是等价的。HEAD~2 代表其第一个父对象的第一个父对象，即祖父对象，而且和 HEAD^^是等价的。

用户还可以对上述内容进行综合应用，例如可以使用 HEAD~3^2 来获得 HEAD 的曾祖父的第 2 个父对象等。用户还可以使用 git name-rev 或者 git describe --contains 命令查找一个修订相关的本地引用，相关代码如下所示：

```
$ git log | git name-rev --stdin
```

## 2.2.5 反向父引用——git 的输出信息描述

父引用描述（describe）记录了当前分支和标签与历史版本的关系。它的内容取决于起

始版本的位置。例如 `HEAD^` 代表的内容和将来的提交截然不同。有时，我们希望描述当前版本和主版本的关系。例如我们希望在生成的二进制应用程序中存放一个可读的当前软件版本名称。然后我们希望该名称的引用指向同一修订并向所有人开放。可以完成这个任务的命令是 `git describe`。

`git describe` 命令会在给定的修订（默认是 `HEAD`）中查找最近的所有标签并使用该修订描述那个版本。如果被找到的标签指针指向了已有的提交，那么（默认情况下）将会只显示该标签。否则，`git describe` 信息中的标签名称会附加标签对象之前的提交数目，以及给定修订的 SHA-1 缩写标识符。例如 `v1.0.4-14-g2414721` 的意思是当前的提交是基于版本 `v1.0.4`（标签）的，在此之前有 14 个提交，`2414721` 是它的 SHA-1 标识符的缩写。

Git 会将这种输出格式当作一种修订声明。

## 2.2.6 reflog 的简称

为了帮助用户从某些错误中恢复，并能够撤销变更（回到状态变更之前），Git 采用了一种叫引用日志（`reflog`）的机制，即存放用户过去数月的 `HEAD` 和分支引用位置以及生成原因的临时日志。引用日志文件默认的有效期限最长可达 90 天，只能通过引用日志访问的修订（例如修订提交）有效期为 30 天。当然这也可以根据引用逐个配置。

用户可以使用 `git reflog` 命令及其子命令查看和编辑引用日志。用户甚至可以使用 `git log -g`（或者使用 `git log --walk-reflog`）命令查看引用日志的历史记录：

```
$ git reflog
ba5807e HEAD@{0}: pull: Merge made by the 'recursive' strategy.
3b16f17 HEAD@{1}: reset: moving to HEAD@{2}
2b953b4 HEAD@{2}: reset: moving to HEAD^
69e0d3d HEAD@{3}: reset: moving to HEAD^^
3b16f17 HEAD@{4}: commit: random.c was too long to type
```

每次用户因故更新 `HEAD` 或者分支首部时，Git 会为用户将这些信息存储在引用历史的本地临时日志中。引用日志中的数据可以用来声明引用（甚至声明修订）：

- 为了声明本地版本库中 `HEAD` 之前的第 `n` 个值，用户可以使用 `HEAD@{n}` 得到和 `git reflog` 命令类似的输出结果。它们的作用都是获得给定分支之前的第 `n` 个值，例如 `master@{n}`。`@{n}` 是个特例，它的含义是获得当前分支之前的第 `n` 个值，这一点和 `HEAD@{n}` 是完全不同的。
- 用户还可以使用这种语法查看分支以往某个时段的情况。例如查找本地版本库

master 分支昨天的情况，可以使用 `master@{yesterday}`。

- 用户还可以使用 `@{-n}` 这样的语法查找当前分支之前的第 `n` 个签出的（工作状态）分支。某些情况下，用户还可以使用 `-` 代替 `@{-1}`，例如 `git checkout -` 会定位到当前分支的上一个分支。

## 2.2.7 上游远程跟踪分支

用户正在工作的项目对应的本地版本库并不是与世隔绝的。它还会和其他版本库交互，至少被它克隆的原生版本库是这样。对于用户经常访问的远程版本库，Git 也会跟踪记录用户最近一次访问的分支节点。

为了记录远程版本库的分支变化，Git 采用的是远程跟踪分支。用户无法在远程跟踪分支上创建新的提交，因为它们有可能在用户下次访问它们时被重写。如果想在远程版本库上的某些分支上提交一些自己的工作成果，那么需要在相关的远程跟踪分支的基础上创建一个本地分支。

例如现在用户工作的名为 `origin` 远程版本库下的 `next` 分支有一系列的功能特性将要发布上线了，那么对应的远程跟踪分支名就是 `origin/next`，系统会在用户本地的版本库中创建一个名为 `next` 的分支。现在我们就说 `origin/next` 分支是 `next` 分支的上游（upstream），并且我们可以使用 `next@{upstream}` 来引用它。后缀 `@{upstream}`（是 `<refname>@{u}` 的缩写）对应的本地分支名是唯一的，分支名的选择和顶部的引用名称前缀有关。如果忽略引用名称，那么系统默认选择当前的分支，即 `@{u}` 代表当前本地分支的上游。

你可以在第 5 章和第 6 章找到远程版本库、上游和远程跟踪分支等概念的详细介绍。

## 2.2.8 根据提交信息查询修订

用户可以通过提交信息的正则表达式查询修订。：`/<模式>` 记号（例如：`/^bugfix`）表示查询任意引用中符合模式的最新提交记录。`<rev>^{<模式>}`（例如 `next^{/fix bug}`）表示从 `<rev>` 引用中查询符合条件的最新提交记录：

```
$ git log 'origin/pu^{/^Merge branch .rs/ref-transactions}'
```

上述修订查询操作也可以使用命令 `git log` 组合 `--grep=<模式>` 参数实现。换句话说，该命令会返回第一个（最新的）符合条件的修订记录，如果只使用 `--grep` 参数，它会返回所有符合条件的修订记录。



## 2.3 修订区间查询

现在我们已经学习了查询单个修订记录的多种方法，接下来看看修订区间查询，即 DAG 子集查询。修订区间对于查询项目某个模块的历史记录非常有用。用户可以使用区间查询回答如下问题：这个分支上还有哪些工作没有和主分支集成？我的主分支上还有哪些功能特性没有发布？这个分支自创建以来都完成了哪些工作？

### 2.3.1 单个修订内部查询

像 `git log` 这类查询历史记录的操作实际的操作是基于组提交的，它会按照从修订的子节点到父节点的顺序遍历所有相关节点。这类命令会把一个给定的修订作为参数（本章的“单个修订内部查询”一节会详细解说该机制），然后遵循提交链回溯至根提交，将所有符合查询条件的提交记录都显示出来。

例如，`git log master` 命令将会显示 `master` 分支（所有该分支上与当前工作相关的修订）相关的所有提交记录，这意味着查询结果会显示整个与 `master` 分支相关的所有工作内容。

### 2.3.2 双点符号

最常用的区间查询声明语法是双点符号，例如 `A..B` 这样一个线性的历史记录，它表示在修订 A 和修订 B 之间的所有记录，甚至还包括更多，例如所有在 B 中但是不在 A 中的提交，参见图 2-3。区间 `HEAD~4..HEAD` 代表 4 个提交：`HEAD`，`HEAD^`，`HEAD^^` 和 `HEAD^^^`。换句话说，即 `HEAD~0`，`HEAD~1`，`HEAD~2`，and `HEAD~3`，当然前提是当前分支和 4 个父提交之间不存在任何合并提交。

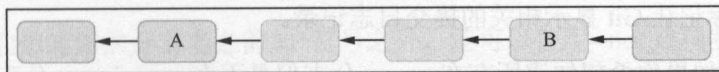


图 2-3 包含双点标记的 `A..B` 线性区间，其范围在上图中是用橘黄色标记的



如果希望将起点提交纳入查询范围（大部分情况下是边界提交），可以将 `git log` 配合 `--boundary` 一起使用，Git 默认情况下会自动忽略该起点提交。



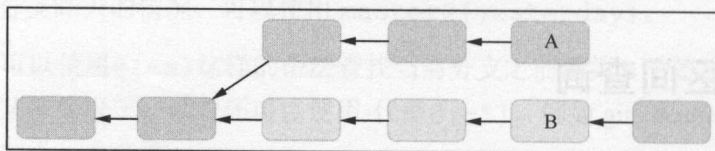


图 2-4 一个包含双点符号的非线性区间 A..B（修订 A 不是修订 B 的祖先提交），

对应的区间用橘黄色表示，不在区间内的节点用阴影表示，

而且边界修订节点还用加粗的边框着重标记

对于非线性历史记录来说情况更复杂一些。当 A 不是 B 的祖先节点（在修订 DAG 中，B 无法回溯 A），但是拥有共同的祖先节点，如图 2-4 所示。非线性历史记录的另外一种情形是 A 和 B 之间存在合并提交，如图 2-5 所示。包含双点标识符的 A..B 或者区间 A 到 B 代表的非线性历史记录更精确的含义，是指包含 A 节点可达但是 B 节点不可达。

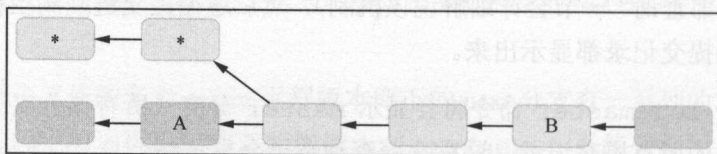


图 2-5 双点标识符表示的非线性历史记录，其中 A 和 B 直接

存在合并提交。排除带\*标记的提交，

可以使用--strict-ancestor 选项

Git 中的 A..B 代表根据继承链关联的一个可达的节点(B)到另外一个不可达的节点(A)之间的一系列提交。如图 2-4 所示，A 和 B 之间的分叉是源自分支点 A 但是只属于 B 的所有提交。

例如你有一个叫 master 的分支，以及其子分支 experiment。你希望知道子分支 experiment 中还有哪些提交没与 master 分支合并，那么可以使用 master..experiment 标记让 Git 显示相关的提交日志记录。

反过来说，如果你希望知道所有在 maser 分支但是不在 experiment 分支中的提交记录，就可以将分支名称反过来表示。experiment..master 标记会显示所有属于 master 分支但是不属于 experiment 分支的提交记录。



又例如 origin/master..HEAD 标记，它会显示用户将要推送到远程版本库的所有提交（当前分支中没有



提交到远程 origin 版本库的 master 分支中的所有提交), HEAD..origin/master 标记会显示用户拉取但未合并的提交。用户还可以省略表达式中的某一边 的值, 让 Git 自动添加 HEAD 引用: origin/master.. 是 origin/master..HEAD 的简写, ..origin/master 是 HEAD..origin/master 的简写。如果其中某一边被省略了, Git 会自动使用 HEAD 补上省略的部分。

Git 中用到双点标记的地方很多, 例如 git fetch 和 git push 命令快进式输出结果中, 可以直接拷贝结果的部分片段作为 git log 的参数。这种情况下, 查询区间的起点是该区间末尾的祖先节点, 而且该区间是线性的:

```
$ git push
To https://git.company.com/random
8c4ceca..493e222    master -> master
```

### 2.3.3 多点符号——包含和排除修订

包含双点符号的 A..B 这样的表达式非常有用并且很直观, 不过实际上它只是一个速记符号。一般情况下它都可以满足需要, 不过有时也许需要外援了。当你需要声明两个以上的分支来标记修订, 例如查看不在当前工作分支下的而是在其他若干分支中的提交记录; 又例如希望查看在 master 分支下的变更, 但是这些变更又不属于其他长线分支的。

Git 允许用户在给定的修订前面加 ^ 前缀将其中的提交排除。例如希望查看 maint 和 master 分支上的所有修订, 但是要排除 next 分支上的, 那么可以使用 git log maint master ^next 命令实现。这说明 A..B 表达式是 B ^A 的快捷方式。

除了在我们想排除的分支名之前加 ^ 符号之外, 还可以使用 --not 选项实现同样的目的, 其后的分支都会被忽略。例如 B ^A ^C 也可以写成 B --not A C。这是一个非常有用的特性, 例如用程序代码排除下列修订时, 下面 3 条命令的作用是等价的:

```
$ git log A..B
$ git log B ^A
$ git log B --not A
```

## 2.3.4 单个修订的修订区间

$A^!$  是另外一个非常有用的快捷方式，它代表一系列单个提交的集合。对于非合并提交集合，它代表  $A^..A$ 。对于合并提交集合， $A^!$  代表的集合会将其所有祖先提交排除。还有一种非常有用的快捷方式是  $A^@$ ，它代表  $A$  的所有祖先提交 ( $A^1, A^2, \dots, A^n$ )。我们可以将  $A^!$  当作  $A --not A^@$  的快捷方式。

## 2.3.5 三点符号

最后一个重要的用于声明修订区间的语法是三点符号，即  $A...B$ 。它表示可以到  $A$  或者  $B$ ，但是不能同时达到两者的提交，如图 2-6 所示。该符号表示  $A$  和  $B$  之间的对称差异。

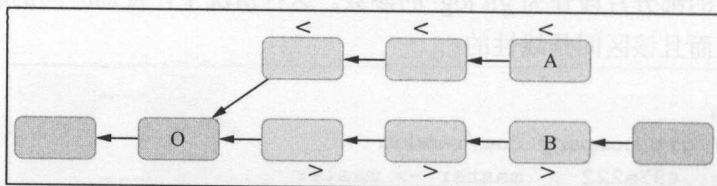


图 2-6 包含三点标识符的  $A...B$  表示一组非线性的历史记录，区间内的元素用橘黄色表示，其边界的提交节点用加粗的黄色边框进行标记。节点元素下方的还采用了 `--left-right` 箭头标记

它是  $A B --not \$(git merge-base --all A B)$  的快捷方式， $\$(...)$  是 shell 命令的占位符（采用 POSIX 的 shell 语法）。这里的意思是 shell 将首先执行 `git merge-base` 命令，找到所有祖先元素（所有之前合并的提交），然后在命令行结果输出中显示其补集。

`git log` 命令在使用三点符号时会结合 `--left-right` 选项一起使用。该选项会显示每个提交的区间两边的记录是否包含在内，使用 `<` 前缀表示坐标的元素在区间内（ $A$  也在区间  $A...B$  中），使用 `>` 表示右边的元素在区间内（ $B$  也在区间  $A...B$  中），如图 2-6 所示。从下面的示例可以看出，使用该选项之后，信息的意义更丰富：

```
$ git log --oneline --left-right 37ec5ed...8cd8cf8
>8cd8cf8 Merge branch 'fc/remote-helper-refmap' into next
>efcd02e Merge branch 'rs/more-starts-with' into next
>831aa30 Merge branch 'jm/api-strbuf-doc' into next
>1aeca19 Merge branch 'jc/revision-dash-count-parsing' into next
<1a7e8e8 Revert "replace: add --graft option"
<7a30690 t9001: avoid non-portable '\n' with sed
>5cc3268 fetch doc: remove "short-cut" section
```



如果`--left-right` 选项和`--boundary` 选项一起使用, 那么它们的含义和忽略边界元素的前缀`-`是等效的。在这种情况下, 对于包含三点符号的 `A...B` 修订区间来说, 它们的边界元素是 `git merge-base --all A B`。

Git 中执行 `git fetch` 和 `git push` 命令属于强制更新时, 其输出结果中会采用三点符号, 这是希望将旧的版本 (左手边) 和更新过的版本 (右手边) 区分开来, 新的版本会强制覆盖旧的版本:

```
$ git fetch
From git://git.kernel.org/pub/scm/git/git
+ 37ec5ed...8cd8cf8 next      -> origin/next (forced update)
+ 9478935...16067c9 pu       -> origin/pu (forced update)
d0b0081...1f58507 todo      -> origin/todo
```



在 `diff` 命令中使用修订区间标识符

为了在执行 `log` 和 `diff` 命令时能够更方便地拷贝和粘贴修订, Git 允许用户使用表示修订区间的双点表达式 `A..B` 和三点表达式 `A...B` 在 `git diff` 命令中指代一组修订 (端点)。

对于 Git 来说, `diff A..B` 命令和 `git diff A B` 命令是等价的, 它们的目的都是找出修订 A 和修订 B 之间的差异。如果可以省略双点标识符两端的修订元素, 那么它的效果和使用 `HEAD` 是一样的。例如 `git diff A..` 和 `git diff A HEAD` 的作用是一样的。 `git diff A...B` 会特意将分支 B 上新增的变更显示出来。新增的变更意味着源自 B 的修订拥有共同的祖先, 即基于 A 和 B 的合并提交。这里 `git diff A...B` 和 `git diff $(git merge-base A B) B` 是等价的, 不过需要注意的是上述 `git merge-base` 中不包含 `--all` 选项。这一约定的结果使得粘贴和复制 `git fetch` 的输出结果 (包括双点和三点表达式) 作为 `git diff` 命令的参数之后, 输出的结果都是已经拉取的变更提交。但是它不包含 A 的子分支上的变更。





此外，这个特性还支持使用 `git diff A^!` 命令查看 A 分支和其父分支的差异（它是 `git diff A^ A` 的快捷方式）。

## 2.4 历史记录查询

`git log` 命令有很多有用的选项参数可供选择，这些选项使得用户能够方便地查看提交记录的子集。在查询修订记录时通过设定修订区间，用户可以查看历史记录的某个特定版本，能够获得比修订 DAG 更有价值的信息。

### 2.4.1 限制修订数量

最常见的方式是限制 `git log` 命令的输出记录数量，同时它也是最简单的限制选项，即只显示最近的提交记录。用户可以使用 `-<n>`（`n` 是任意整数）选项做到这一点，该选项还可以写成 `-n <n>`，或者写成最大天数的形式 `--max-count=<n>`。例如 `git log -2` 将会显示当前分支上最近两天的工作提交记录，而且起始点是 `HEAD` 指向的修订。

用户还可以使用 `--skip=<n>` 选项忽略最近的部分提交记录。

### 2.4.2 元数据查询

历史记录限制选项可以分为存储在提交对象内部的信息校验，以及在变更集（父提交上的变更）上的提交内容过滤。

#### 时间段查询

如果用户希望查询过去某个时间段的提交记录，可以使用诸如 `--since`、`--until`、`--before` 和 `--after` 选项。例如，下面的命令会显示最近两周的提交记录：

```
$ git log --since=2.weeks
```

这些选项支持多种格式。用户可以声明一个特定的日期，例如 `2008-04-21`，或者一个相对日期，例如 `2 年`、`3 个月` 和 `3 天前`；用户还可以使用点标识符代替空格。

当使用一个特定日期时，如果该日期没有指定时区，用户需要留意系统会默认使用当地的时区。这一点非常重要，因为在这种情况下，当团队成员分布于全球各地不同时区时，

Git 系统查询到的结果将无法保证一致性。例如`--since="2014-04-29 12:00:00"`选项将会导致 Birmingham, England, United Kingdom (对应的标准时区为 2014-04-29Z11:00:00) 比 Birmingham, Alabama, USA (对应的标准时区为 2014-04-29Z17:00:00) 早 6 小时。为了让大家得到一样的结果, 需要在时间选项中明确指定时区, 例如`--after="2013-04-29T17:07:22+0200"`。

另外需要注意, Git 系统中包含两个用于描述修订的时间——作者日期和提交者日期。时间限制选项在这里主要是和提交者时间相关的, 即创建修订对象的时间和日期。这一点和作者日期有一些差别, 作者日期指的是一组变更集被创建 (产生变更时) 的时间和日期。

个别情况下作者信息包含的日期可以和提交者信息的日期不同。例如当提交在一个版本库中创建后, 修改了电子邮件信息之后, 某人又将之应用到另外一个版本库中。日期不同的另外一种情况是执行变基操作时提交的重新创建, 默认情况下, 它会保留作者日期, 生成一个新的提交者日期 (详情可以参考第 8 章)。

## 提交内容查询

如果只希望查看某个作者或者提交者的历史修订记录, 可以分别使用`--author` 或 `--committer` 选项。例如希望查看源代码版本库中名为 Linus 的作者的提交记录, 可以使用命令 `git log --author=Linus` 实现。一般来说, 该搜索是大小写敏感的, 并且支持正则表达式。Git 会搜索提交作者的名字和电子邮件中包含查询提交的记录; 或者只匹配作者名字可以使用`--author=^Linus` 来实现。

`--grep` 选项可以帮助用户搜索提交信息 (变更记录的描述信息)。例如用户希望找到所有在提交信息中使用通用漏洞标识符标记的安全 bug 修复提交记录, 那么用户可以 `git log --grep=CVE` 命令达到上述目的。

如果在命令中同时声明了`--author` 和 `--grep` 两个选项, 或者声明了多个`--author` 或者 `--grep` 选项, Git 系统将会把同时符合上述条件的结果显示出来。换句话说, Git 会对符合条件的提交记录进行逻辑或操作。如果用户希望获得所有符合条件的提交记录, 即执行逻辑与操作, 则需要使用`--all-match` 选项。

还有一组选项用于指定匹配模式, 它们和 `grep` 程序中采用的方式类似。为了让搜索支持大小写敏感, 可以使用`-i / --regexp-ignore-case` 选项。如果用户希望进行子字符串匹配, 可以使用`-F / --fixed-strings` 选项 (例如有时需要在正则表达式中诸如.和?等符号进行转义)。为了构建更强大的查询语句, 用户还可以使用`--extended-`

regex 和 `--perl-regex` 等选项进行功能扩展（使用最新的 Git 程序，并且需要支持 PCRE 库）。

## 父提交

一般来说，Git 系统允许用户沿着继承链追溯每个合并提交的父对象。例如只希望查看当前元素的第一个父对象，那么可以使用相应的 `--first-parent` 选项。如果遵循了合并变更的特定做法，那么这将会为用户显示主线分支（有时也称主干）的一系列历史提交记录。第 7 章将会为读者深入介绍相关知识。

比较如下代码（该示例巧妙地展示了使用 `--graph` 选项后，用 `ascii` 码直观构造的历史记录查询结果）：

```
$ git log -5 --graph --oneline
* 50f84e3 Update draft release notes to 2.1
* 07768e0 Merge branch 'jc/shortlog-ref-exclude'
|\
| * eb07774 shortlog: allow --exclude=<glob> to be passed
* | 251cb96 Merge branch 'mn/sideband-no-ansi'
|\ \
| * | 38de156 sideband.c: do not use ANSI control sequence
```

以及下面的代码：

```
$ git log -5 --graph --oneline --first-parent
* 50f84e3 Update draft release notes to 2.1
* 07768e0 Merge branch 'jc/shortlog-ref-exclude'
* 251cb96 Merge branch 'mn/sideband-no-ansi'
* d37e8c5 Merge branch 'rs/mailinfo-header-cmp'
* 53b4d83 Merge branch 'pb/trim-trailing-spaces'
```

用户可以使用 `--merges` 选项只显示列表中的合并提交，使用 `--no-merges` 选项只显示列表中的非合并提交。上述选项是另外一个更通用的选项的快捷方式，即 `--min-parents=<number>`（`--merges` 用 `--min-parents=2` 表示）和 `--max-parents=<number>`（`--no-merges` 用 `--max-parents=1` 表示）。

例如用户希望查看项目的起点，那么可以使用 `--max-parents=0` 选项，它会把所有根提交显示出来：

```
$ git log --max-parents=0 --oneline
0ca71b3 basic options parsing and whatnot.
```



```

16d6b8a Initial import of a python script...
cb07fc2 git-gui: Initial revision.
161332a first working version
1db95b0 Add initial version of gitk to the CVS repository
2744b23 Start of early patch applicator tools for git.
e83c516 Initial revision of "git", the information manager from hell

```

## 2.4.3 修订内部变更查询

有时查询提交信息和修订元数据不一定能够满足实际需要。也许变更的描述信息不够详尽，也有可能正在查找的修订中功能函数已经被其他模块调用，甚至发生变量重名的情况。

Git 允许用户对每个修订内部的变更进行查询（比较提交和其祖先提交的差异），一般称之为 pickaxe 搜索。

通过 `-S<string>` 选项，Git 将会查找给定字符串实例的引用或者移除产生的差异。注意这一点和 `diff` 命令的输出结果完全不同（还可以使用 `--pickaxe-regex` 选项进行正则表达式匹配搜索）。Git 系统检查每个修订对应的提交记录，如果文件中当前提交和父提交中包含特定字符串形式的不同数字，将会显示符合条件的修订记录。

不过 `git log` 搭配 `-S` 选项也会显示每个修订的变更记录（效果和使用 `--patch` 类似），不过它只显示符合条件的差异记录。为了显示所有文件的差异，而不是只显示编号变更的差异，用户需要使用 `--pickaxe-all` 选项：

```

$ git log -S'sub href'
commit 06a9d86b49b826562e2b12b5c7e831e20b8f7dce
Author: Martin Waitz <tali@admingilde.org>
Date:   Wed Aug 16 00:23:50 2006 +0200

```

gitweb: provide function to format the URL for an action link.

Provide a new function which can be used to generate an URL for the CGI.

This makes it possible to consolidate the URL generation in order to make

it easier to change the encoding of actions into URLs.

Signed-off-by: Martin Waitz <tali@admingilde.org>

Signed-off-by: Junio C Hamano <junkio@cox.net>



使用-G<regex>选项, Git 将会逐字逐句地在添加或者删除的提交记录中根据给定的正则表达式查找符合条件的差异。需要注意的是。统一的 diff 格式 (Git 采用的) 将会把一系列的变更记录表示成移除旧版本和添加新版本的形式; 第 3 章将会详细介绍 Git 如何描述变更记录。为了区分-S<regex>、--pickaxe-regex 和-G<regex>三者之间的差异, 可以通过使用 diff 命令查看同一文件的提交记录的使用来区分:

```
if (lstat(path, &st))
-     return error("cannot stat '%s': %s", path,
+     ret = error("cannot stat '%s': %s", path,
                  strerror(errno));
```

git log -G"error\"(" 命令将会显示这个提交 (因为查询条件匹配), git log -S"error\"(" --pickaxe-regex 则不会 (因为字符串出现的次数没变化)。



如果只对单个文件感兴趣, 可以使用 git blame (也可以使用图形式的浏览器, 例如 git gui blame) 方便地查看已经引入的变更记录。

不过 git blame 命令不能查看已经被删除的提交记录, 要实现该功能, 需要使用 pickaxe 搜索才能达到目的。

## 2.4.4 变更类型查询

有时用户也许只是想看看文件增加和删除的情况, 那么在 Git 中, 使用 log --diff-filter=AM 命令即可。用户可以查询任意类型的变更; 用户手册 git-log(1) 章节有详细的介绍说明。

## 2.5 单个文件历史记录

如本章前面介绍工作区目录提交的相关章节所述, Git 的修订是将整个项目的状态当作一个独立的实体。

多数情况下, 特别是大型项目中, 开发人员一般只对单个文件的历史版本或者某一个给定目录下 (给定的子系统下) 的文件变更感兴趣。

## 2.5.1 路径约束

为了查看单个文件的历史版本，用户可以使用 `git log <路径名>`。Git 将会显示给定路径（一个文件或者文件夹）下的所有修订历史记录，这些修订记录可能是一个给定文件的变更，也可能是一个子目录中的某个文件的变更。

### 消除分支和路径名之间的歧义



Git 在用户输入 `git log foo` 命令时，会猜测用户的意图，例如是否是希望查看分支 `foo` 的历史记录，又或者是文件 `foo` 的历史记录。不过，有时候 Git 系统会变得无从选择。为了防止文件路径名和分支名发生混淆，可以使用 `--` 将文件路径名和其他选项区分开来。任何在 `--` 后的字符串都将被当作文件名，在其之前的选项被当作分支名或者其他选项。

例如 `git log -- foo` 命令就是明确地告诉 Git 查看路径名为 `foo` 的历史记录。

除了分支名和文件名重复之外，另外一种常见的情况是查询一个已删除文件的历史记录，即该文件已经不在项目中了。

用户可以声明多个路径，甚至可以根据文件类型进行通配符查询（模式匹配）。例如只想找到 Perl 脚本文件（该文件的后缀名为 `.pl`）中的变更，可以使用 `git log -- '*.pl'`。不过需要注意，务必确保在 Git 系统查看该文件之前，shell 可以正确识别 `*.pl` 通配符，例如上述通配符中的点标记。

Git 在显示项目历史记录时是使用路径名作为条件参数的，查询单个文件的历史版本时系统并不会自动进行重命名。用户需要使用 `git log --follow <文件>` 来避免重名的问题。但是不幸的是，上述命令并不是在任何场景都能奏效。有时用户需要使用 `blame` 命令（详情见下一节），并且在查看边界提交时启用重命名检测功能，然后手动执行文件的重命名和移动操作。

在当前的 Git 系统中，用户还可以使用 `git log -L` 命令查看单个文件内部的历史演变记录，即对单个文件的某个修订（可以指定修订编号）进行查看。修订区间可以通过 `-L`

<start>, <end>:<file>参数进行具体指定 (<start>和<end>参数可以是数字或正则表达式), 也可以使用一个功能函数进行追踪, 即-L :<funcname regexp>:<file>。不过上述参数不能和规范的路径约束条件一起使用。

## 2.5.2 历史简化

一般来说, 在查询路径历史记录时, Git 系统会对历史记录进行概要显示, 只显示被查询的提交相关文件对应的特定路径。Git 将会排除对给定文件没有影响的修订记录。此外, 对于未排除的合并提交记录, Git 会排除这些提交的父提交记录 (排除不相关的功能特性)。

用户为了简化这些历史记录, 可以使用 `git log` 命令配合 `--full-history` 或者 `--simplify-merges` 选项一块使用。可以查阅 Git 的帮助文档了解具体细节, 例如帮助文档中 `git-log(1)` 的 "History Simplification" 章节。

## 2.5.3 blame——查看文件历史记录详情

`blame` 命令是一个版本控制特性, 专门用于帮助用户查看文件编辑者的信息。该命令可以查看文件内部每一行内容的变动, 例如新增一行、文件的编辑者等。它会找到最新的提交记录对应新增的每一行内容。一个已经提交的修订对应的每一行内容和其对应的父修订在形式上是完全不同的。`git blame` 命令默认的输出信息中每一行内容都包含相应的编辑者信息。

Git 系统可以为给定的修订添加附注信息 (这对于浏览文件历史和比较文件差异来说非常有用), 甚至可以为给定的修订区间进行查询。用户甚至可以限制一系列的附注内容范围使得文件审核更快, 例如用户只希望检查 `gitweb/ gitweb.perl` 文件中的 `esc_html` 函数, 那么可以执行下列命令:

```
$ git blame -L '/^sub esc_html {/,/}' gitweb/gitweb.perl
```

`blame` 命令非常有用, 它甚至可以追踪文件重命名的整个历史记录。它还可以任意地追踪文件内容从一个文件移动到另外一个文件的记录 (使用 `-M` 选项), 以及文件之间的拷贝、粘贴操作 (使用 `-C` 选项), 甚至文件内部的代码移动。

在跟踪代码移动记录时, 忽略空格是非常有用的, 这可以帮助用户识别代码段是新增的还是因为重新排版 (例如重构代码造成的代码段删减) 而发生的变更。上述目的可以通过 `diff` 命令的格式化选项 `-w` 或者 `--ignore-all-space` 实现。



### 重命名监测

好的版本控制系统应该能够处理文件重命名和项目结构发生变化这类问题。有两种方式处理上述问题。第一种是重命名跟踪,这意味着在提交变更时,会将保存了文件重命名相关操作的记录信息一并提交,即版本控制系统会对重命名的文件进行标记。重命名文件时一般需要用到 `rename` 和 `move` 命令(没有使用非版本控制的文件管理器),还可以在创建修订时对文件进行重命名监测。整个重命名过程也是文件特征存续的过程。

第二种方法,也是 Git 比较常见的方式,即重命名监测。在这种情况下, `mv` 命令只是删除一个旧的文件(包括原有的文件名),然后添加一个新的文件(包括新的文件名)的快捷方式。重命名监测实际上意味着需要对文件重命名进行监测的时候:当执行合并,查看文件历史内容(如果必要的话),或者执行 `diff` 命令时(如果有必要或者已经配置)。这样做的好处是可以改进重命名监测算法,而且同时在提交时不会被冻结。这是一种更通用的解决方案,不仅可以处理整个文件的重命名,而且单个文件内部和多个文件之间的代码移动和粘贴操作也能在 `git blame` 命令的描述信息中显示出来。

重命名监测的缺点在于, Git 系统基于文件内容和路径的启发式相似解析,它需要占用一定的资源,极个别情况下可能会失败,无法监测重命名或者被监测信息并不存在。

注意,在 Git 中, `diff` 命令中默认是没有启用重命名监测功能的。



Git 中的 `blame` 命令包含很多对应的图像化操作接口。`git gui blame` 命令就是 `blame` 操作的图形化示例应用(它是基于 Tcl/Tk 图形化接口的 git 图形化应用之一)。这类图形化接口可以显示变更的全部描述信息,同时还可以显示相关的文件重命名历史记录。通过这类图



形化接口应用程序，用户可以查找某个特定的提交记录，交互式浏览文件历史记录。此外，如图 2-7 所示，blame 的图形化接口 (GUI) 应用可以方便地追踪文件重命名的整个流程记录。

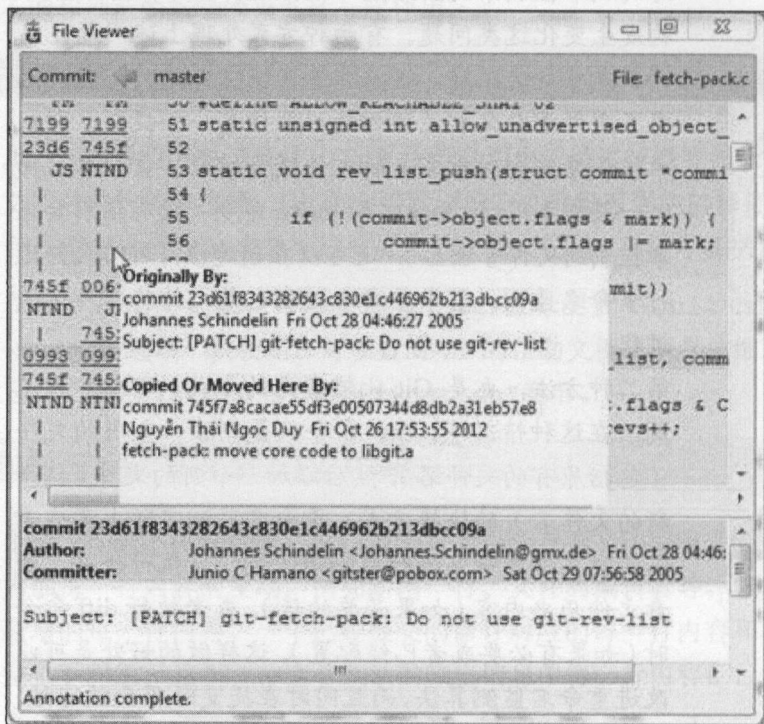


图 2-7 blame 的 GUI 应用，展示了粘贴和复制代码片段的操作历史记录

## 2.6 使用 git bisect 命令查找 bug

Git 为用户提供了两种工具帮助用户查找项目的问题。这些工具通常都是非常有用的，尤其是在软件回归检查过程中，在特定修订版本中给软件加入新特性之后可能会产生 bug。如果不知道 bug 在哪里，那么这个修订中可能存在成千上百的提交记录，这时 git bisect 命令就可以大显身手了。

bisect 命令会半自动化地一步一步搜索项目历史，尝试找出产生 bug 的修订版本。在执行每个步骤过程中，项目历史会被分成大致相等的两个部分，然后分别对其中的变更记录进行查询，看看其中是否存在 bug (二分查找)；然后系统会根据这种办法过滤两个部分中的一个，继而减小了产生 bug 的变更的修订区间。

例如 1.14 版本的程序能够正常运行，但是 1.15-rc0 预览版程序有时会发生崩溃。那

么可以将程序回退到 1.15-rc0，然后确保可以在该版程序中重现该问题（这一点非常重要），不过不知道导致该问题的原因是什么。

用户可以对代码历史进行二分查找，对问题进行定位。首先使用 `git bisect start` 命令启动查找过程，然后使用 `git bisect bad` 命令告诉 Git 哪个版本存在问题，最后使用 `git bisect good` 命令告诉 Git 系统最新的稳定程序版本号：

```
$ git bisect start
$ git bisect bad v1.15-rc0
$ git bisect good v1.14
Bisecting: 159 revisions left to test after this (roughly 7 steps)
[7ea60c15cc98ab586aea77c256934acd438c7f95] Merge branch 'mergetool'
```

从上述信息可知，Git 系统告知用户在稳定版程序（v1.14）和有问题的版本（v1.15-rc0）之间大约有 300 个变更记录，而且它还为用户签出了最中间的那条变更记录（7ea60c15）。如果用户现在执行 `git branch` 命令，那么将会看到系统已经临时将用户移动到了签出的那条变更记录下（无分支）：

```
$ git branch
* (no branch, bisect started on master)
master
```

接下来用户需要做的是运行测试程序，看看通过 `bisect` 命令签出的变更记录是否存在问题。如果经过测试发现程序崩溃了，那么使用 `git bisect bad` 命令对该变更记录进行标记；如果发现该变更没什么问题，那么可以使用 `git bisect good` 命令对其进行标记。经过大概 7 个循环往复的步骤之后，Git 将会定位到可疑的变更记录：

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800
```

```
secure this thing
```

```
:040000 040000 40ee3e7... f24d3c6... M config
```

上述示例中的最后一行是原始的 `diff` 命令输出结果，它会显示提交记录中相关的文件变更记录。用户可以使用 `git show` 命令查看可疑的变更记录。这样一来用户就可以找到提交上述记录的开发人员，然后咨询他们或者要求他们修复该问题（给他们发送问题报告）。

如果在项目开发过程中都遵循良好的小步增量变更迭代，那么查找有问题的代码变更记录的数量也会非常少。

当用户发现某个变更记录和其他记录关联不大，而且不好测试，那么可以使用 `git bisect skip` 命令忽略它，甚至还可以使用 `skip` 子命令对给定的修订区间忽略一系列的变更记录。当执行上述检查工作后，用户应该执行 `git bisect reset` 命令返回分支的起始点。

```
$ git bisect reset
Previous HEAD position was b047b02... secure this thing
Switched to branch 'master'
```

为了完成二分检查之后定位到有问题的变更记录，用户可以使用 `git bisect reset HEAD` 命令。

用户还可以使用 `git bisect run` 命令实现查找问题的完全自动化。为此，用户需要提供一个脚本来测试变更记录是否存在问题，并且使用 0 作为返回值表示整个项目一切正常，或者使用一个非 0 数字来提示用户存在问题。这里 125 作为特殊的返回代码，表示当前签出的变更记录程序无法测试。首先，用户需要为 `bisect` 操作声明正常提交记录和有问题的提交记录的区间。可以根据需要使用 `bisect start` 命令查看这些记录，先列出有问题的版本号，然后再显示正常的版本号。用户如果知道受到影响的目录结构，那么还可以声明路径参数，有效减少排查问题的次数：

```
$ git bisect start v1.5-rc0 v1.4 -- arch/i386
$ git bisect run ./test-error.sh
```

对每个签出的提交自动执行 `test-error.sh` 这样一个测试脚本，直到 Git 查到问题所在。这里我们已经使用 `bisect start` 指定了好的提交版本和坏的提交版本记录，先显示有问题的提交记录，然后显示正常的提交记录。

如果项目的 bug 导致了编译停止（编译失败），那么可以使用 `make` 命令作为测试脚本（`git bisect run make`）。

## 2.7 日志的查询和格式化输出

目前已经介绍了如何查询修订记录和对查询结果进行过滤（只搜索感兴趣的部分），接下来要介绍如何显示查询修订结果，以及格式化查询结果。`git log` 命令包含大量的选项参



数来达到上述目的。

## 2.7.1 预定义和用户自定义输出格式

--pretty 是 git log 命令最有用的可选参数之一。该选项可以修改日志输出的格式。Git 系统内置了不少预定义的格式供用户选择。在线格式可以把每个提交单独打印为一行输出，这对于用户查看大量提交记录时非常有用，事实上 --oneline 选项就是 --pretty=oneline --abbrev=组合的快捷方式。此外，短的、适中的（默认）、长的和超长的格式输出本质上是一种格式，区别只在于它们包含的信息量的多少。提交记录在 Git 内部采用的都是原生格式。

在这些冗长的格式中可以使用 --date 选项修改其中的日期格式：使用 -date =relative 选项，让 Git 显示相对日期，例如两小时前。使用 --date=local 选项可以显示本地时区的时间格式等。

用户还可以使用 --pretty=format:"<string>"（还有其变种 tformat）选项配置自己的日志格式。这在用户对用脚本生成的输出结果进行机器解析时非常有用，因为当用户显式声明格式后，Git 系统不会对它更新修改。格式化字符串的工作原理和 printf 语句类似：

```
$ git log --pretty="%h - %an, %ar : %s"
50f84e3 - Junio C Hamano, 7 days ago : Update draft release notes
0953113 - Junio C Hamano, 10 days ago : Second batch for 2.1
afa53fe - Nick Alcock, 2 weeks ago : t5538: move http push tests out
```

表 2-1 列出了很多格式占位符供用户选择。

表 2-1 占位符及含义

占 位 符	含 义
%H	提交的哈希码（修订的 SHA-1 完整标记）
%h	提交哈希码简称
%an	作者名称
%ae	作者电子邮件
%ar	作者相对日期
%cn	提交者姓名



续表

占 位 符	含 义
%ce	提交者电子邮件
%cr	提交者相对日期
%s	主题（提交信息的第一行，用户对修订的描述）
%%	百分比符号（%）



### 作者和提交者的区别

作者是创建变更的人，而提交者是在原作者创建的变更的基础上进一步优化该变更的人（创建包含若干变更的提交对象，在 DAG 图中用修订表示）。因此，如果你给项目提交了一个修订，某个核心开发人员又完善了该修订，那么你就是作者，那个核心开发人员就是提交者。

--graph 选项虽然可以和其他任意格式组合使用，但是它与--oneline 格式选项在 git log 命令中搭配使用效果特别好。前者可以为输出结果添加由 ASCII 图形显示的分支和合并记录历史。如果还希望查看标签和分支的构成，还可以添加上--decorate 选项：

```
$ git log --graph --decorate --oneline origin/maint
* bce14aa (origin/maint) Sync with 1.9.4
|\
| * 34d5217 (tag: v1.9.4) Git 1.9.4
| * 12188a8 Merge branch 'rh/prompt' into maint
| |\
| * \ 64d8c31 Merge branch 'mw/symlinks' into maint
| |\ \
* | | | d717282 t5537: re-drop http tests
* | | | e156455 (tag: v2.0.0) Git 2.0
```

也许用户希望使用一种图形化工具来查看自己的提交记录。类似的软件是一个 Tcl/Tk 程序，它的名字叫 gitk，并且是由 Git 团队发布的。你可以在第 10 章了解到上述图形化工具的更多介绍信息。

## 2.7.2 包含、格式化和统计变更

用户可以使用 git show 命令查看单个修订记录以及提交的元数据，并且使用统一的 diff

格式显示上述查询结果。不过有时用户也许希望查看 `git log` 命令输出结果中的部分历史记录。这时用户可以使用 `-p` 选项达到此目的，或者快速浏览团队成员新增的提交记录。

一般来说，Git 不会显示合并提交的变更记录。如果希望查看所有父提交的记录，用户可以使用 `-c` 选项（或者使用 `-cc` 选项显示压缩过的输出记录），也可以使用 `-m` 选项单独显示某个变更的父提交。

`git log` 支持使用多个选项来改变 `diff` 输出的格式。有时查看单词级别的变更要比行级的更容易一些。`--word-diff` 就是上述选项之一。它对于比较文档文件内部的变更来说非常有用（例如帮助文档）：

```
commit 06ab60c06606613f238f3154cb27cb22d9723967
Author: Jason St. John <jstjohn@purdue.edu>
Date:   Wed May 21 14:52:26 2014 -0400
```

```
Documentation: use "command-line" when used as a compound adjective,
and fix
```

```
Signed-off-by: Jason St. John <jstjohn@purdue.edu>
```

```
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

```
diff --git a/Documentation/config.txt b/Documentation/config.txt
index 1932e9b..553b300 100644
```

```
--- a/Documentation/config.txt
```

```
+++ b/Documentation/config.txt
```

```
@@ -381,7 +381,7
```

```
Set the path to the root of the working tree.
```

```
This can be overridden by the GIT_WORK_TREE environment
```

```
variable and the '--work-tree' [-command line-]{+command-line+}
```

option.

```
The value can be an absolute path or relative to the path to
the .git directory, which is either specified by --git-dir
or GIT_DIR, or automatically discovered.
```

另外一组非常有用的选项是和忽略空格变更有关的，其中 `-w / --ignore-all-space` 选项可以忽略所有空格变更，`-b / --ignore-space-change` 选项可以忽略一定数量的空格变更。

有时，用户也许只对变更摘要感兴趣而非具体细节，因此也有一系列的 `diff` 摘要选项

供用户选择。如果用户只想知道哪些文件发生变更了，那么可以使用`--names-only` (or `--raw --abbrev`) 选项。如果用户还希望知道发生变更的文件数量，那么可以使用`--stat` 选项（或者使用对机器更友好的`--numstat` 选项）。如果用户只对变更摘要感兴趣，那么可以使用`--shortstat` 或者`--summary` 选项。

### 2.7.3 贡献统计

你可曾想要看看自己已经为项目贡献了多少提交记录？又或者上个月项目组最活跃的开发者是谁（从贡献数目来说）？那么接下来要介绍的 `git shortlog` 命令是专门为此而生的：

```
$ git shortlog -s -n
13885 Junio C Hamano
1399 Shawn O. Pearce
1384 Jeff King
1108 Linus Torvalds
743 Jonathan Nieder
```

`-s` 选项将会把所有提交信息表示为提交数目，如果没有使用该选项的话，`git shortlog` 将会显示所有提交的摘要信息，并根据开发者进行分组（也可以使用类似`--format` 选项进行配置优化、输出结果）。`-n` 选项可以根据开发者提交的修订数量排序，此外还可以根据开发者姓氏排序。用户还可以添加`-e` 选项显示开发人员的电子邮件，不过需要注意的是，采用该选项之后，Git 系统会对使用多个电子邮件的同一作者提交的修订记录进行重新分组。

`git shortlog` 命令还可以接受一个修订区间作为参数，以及其他修订区间限制选项，例如`--since=1.month.ago`。`git log` 命令可以使用的参数也几乎适用于 `shortlog`。例如，用户希望查看项目最新的预览版程序的开发人员，可以使用如下命令：

```
$ git shortlog -e v2.0.0-rc2..v2.0.0-rc3
Jonathan Nieder <jrnieder@gmail.com> (1):
    shell doc: remove stray "+" in example
```

```
Junio C Hamano <gitster@pobox.com> (14):
Merge branch 'cl/p4-use-diff-tree'
Update draft release notes for 2.0
Merge branch 'km/avoid-cp-a' into maint
```



需要注意的是，统计作者的修订数量只是衡量项目贡献大小的方法之一。例如某些开发人员提交的专门用于 bug 修复的修订数量，可能远远大于没有引入 bug 的开发者（或者是软件发布前的问题修复）。

还有其他专门衡量程序员生成效率的指标，例如提交记录中发生变更的行数，或者可以通过 Git 系统统计的现有行数，不过系统没有内置的命令可以用于统计这些数据。

### 作者映射

对于项目周期比较长的 Git 版本库来说，执行 `git shortlog -s -n -e` 或者 `git blame` 命令时可能会遇到的问题是，开发人员在推进项目的过程中会变更他们的名字和电子邮件，或者两者兼而有之。其原因有很多：更换工作（工作电子邮件也随之变更），错误的配置，拼写错误等：

```
Bob Hacker <bob@example.com>
```

```
Bob <bob@example.com>
```



当发生上述情况时，用户就无法获得正确的属性信息了。Git 允许用户在项目顶层目录中的 `.mailmap` 文件中配置 `author/e-mail` 的信息对。它允许用户为开发人员指定标准信息，例如：

```
Bob Hacker <bob@example.com>
```

（实际上，它允许用户声明标准的姓名、电子邮件或者兼而有之，以便对电子邮件或者姓名进行精确匹配。）

默认情况下，这些修正信息将会对 `git blame` 和 `git shortlog` 命令产生影响，但是不会影响 `git log` 命令的输出结果。当用户在自定义输出结果样式时，可以使用占位符对输出结果中的姓名和电子邮件进行修正，或者使用 `--use-mailmap` 选项，还可以使用 `log.mailmap` 文件中的配置变量。

## 2.7.4 查看文件修订

有时，用户也许会希望查看单个修订的细节（例如使用 `git bisect` 定位某个疑似有问题



的提交记录), 以及相关变更的描述信息。又或者希望查看某个提交相关的附注标签对应的标签信息。Git 专门提供了一个通用的命令 `git show` 来做这些事情, 而且它可以用来查看任何类型的对象。

例如, 希望查看当前修订版本的祖先提交, 那么可以使用如下命令:

```
$ git show HEAD^^
commit ca3cdd6bb3fcd0c162a690d5383bdb8e8144b0d2
Author: Bob Hacker <bob@virtech.com>
Date: Sun Jun 1 02:36:32 2014 +0200
```

```
Added COPYRIGHT
```

```
diff --git a/COPYRIGHT b/COPYRIGHT
new file mode 100644
index 0000000..862aafd
--- /dev/null
+++ b/COPYRIGHT
@@ -0,0 +1,2 @@
+Copyright (c) 2014 VirTech Inc.
+All Rights Reserved
```

`git show` 命令还可以用来显示目录(树视图)和文件内容(blob 对象)。为了浏览一个文件(或者目录), 用户需要指定目标的版本信息和路径, 然后使用 `:` 将它们连接到一起。例如要查看文件 `src/rand.c` 的内容, 以及该文件对应的标签 `v0.1`:

```
$ git show v0.1:src/rand.c
```

上述做法可能比 `git checkout v0.1 -- src/rand.c` 这样的从工作目录将特定版本的文件签出更方便一些。在冒号之前可以是和提交记录相关的任何名称, 其后也可以是被 Git 跟踪的任何文件(这里的文件是 `src/rand.c`)。这里的路径名是包含顶层目录的文件路径, 不过用户还可以在冒号后面使用相对路径`.`, 例如用户当前的位置是在 `src/`子目录下, 那么可以使用 `v0.1:./rand.c`。

用户可以使用同样的技巧比较任意文件的任意修订记录。

## 2.8 小结

本章主要介绍了几种常用的查看项目历史的方法: 查找相关的修订记录, 修订的查询

和过滤显示，以及格式化输出结果。

首先介绍了项目历史的概念模型——修订的有向无环图（DAG）。理解该概念是非常重要的，因为很多查询工具直接或者间接地和 DAG 有关。接下来介绍了查询单个修订和修订区间查询。我们可以使用上述知识了解某个分支与主分支发生分离时发生了哪些变更，以及找到和某个开发人员相关的所有修订。

我们甚至可以在浏览代码的历史记录时对 bug 进行定位：使用 `pickaxe` 搜索查询某个函数是何时从代码文件中删除的，使用 `git blame` 命令查看某个文件的代码演变历史以及编辑该代码的开发人员，还可以使用 `git bisect` 命令通过半自动或者全自动方式搜索项目历史，以便找到可能引入问题的修订版本。

当查看某个修订时，用户可以选择查询结果的信息显示格式，甚至可以使用用户自定义格式。有多种方式显示信息摘要，其中包括文件变更记录统计和每个开发人员的提交记录数目。

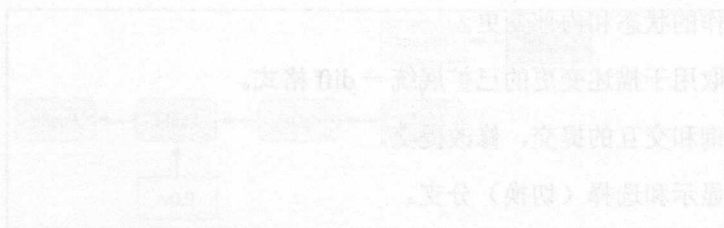


图 1-1 项目历史的概念模型——修订的有向无环图（DAG）

HEAD 指向当前的修订版本是 `branch`，它指向的修订版本是 `branch`。

Git 系统会移动 `master` 分支指向最新的提交（即分支的 DAG 图中最新的提交）。你可以看到，新的提交是用加粗的红色标记了出来，`master` 分支上的提交是用普通的方式表示。需要记住的是，HEAD 指向并没有改变，它一直是指向 `branch`。

新的提交 `branch` 是用加粗的红色标记的，`master` 分支上指向的提交是用普通的方式表示。需要记住的是，HEAD 指向并没有改变，它一直是指向 `branch`。

## 第 3 章

# 使用 Git 进行程序开发

上一章介绍了如何查看项目历史，本章将会介绍如何创建和添加上述历史记录。我们将学习如何创建新的修订和开发特性，还会介绍在开发过程中如何利用 Git 提高工作效率。

本章将会重点介绍个人开发提交记录的主要流程。团队协作开发的内容会在第 5 章进行详细介绍，同时第 7 章会介绍如何利用 Git 对项目进行日常维护。

本章还会介绍 Git 中非常重要的概念，即暂存区（索引），以及脱离 HEAD 的匿名分支。读者还会了解用于描述变更的已扩展统一 diff 格式。

下面的列表是本章将要介绍的主要内容。

- 索引- 提交的暂存区。
- 查看工作的状态和内部变更。
- 如何读取用于描述变更的已扩展统一 diff 格式。
- 支持查询和交互的提交，修改提交。
- 创建、显示和选择（切换）分支。
- 切换分支失败的原因，以及应对策略。
- 使用 `git reset` 对分支复位。
- 与 HEAD 脱离的匿名分支（例如签出的标签等）。

### 3.1 新建提交

在使用 Git 开始一个新项目之前，用户需要如第 1 章所述，先添加自己的姓名和电子

邮件信息。上述信息主要用于标记开发人员的工作记录，无论作者还是提交者都是如此。该设置可以是适用于所有版本库的全局配置（使用 `git config --global` 命令，或者直接编辑 `~/.gitconfig` 文件），也可以是仅局限于用户本地版本库（使用 `git config` 命令或者编辑 `.git/config` 文件）。每个版本库的配置将会覆盖单个开发人员的配置（详情可以参考第 10 章的内容）。某些用户也许会在公司的版本库下工作时使用工作电子邮件，在某些公共版本库中使用非工作电子邮件。

相应的配置文件信息与下列内容类似：

```
[user]
name = Joe R. Hacker
email = joe@company.com
```

### 3.1.1 新建提交的 DAG 视图

第 2 章已经介绍过修订的有向无环图（DAG）这一概念。为一个项目做贡献通常意味着为上述项目添加新的修订，然后将它们作为提交节点添加到修订视图上。

接下来假定我们现在处于 `master` 分支，如图 3-1 所示，然后我们想为项目添加一个新的修订版本（具体操作将会在后续章节详细介绍）。`git commit` 命令将会创建一个新的提交对象——一个新的修订节点。该提交会创建一个特定的签出修订版本（本示例是 `c7cd3`）。该修订会被追踪引用的起点指针 `HEAD` 引用，即当前图表中 `HEAD` 指向的节点 `c7cd3`。

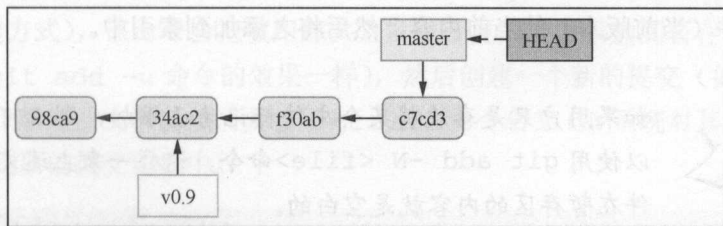


图 3-1 项目启动之初并且没有新增提交的修订视图（DAG）。当前的分支是 `master`，

`HEAD` 指针指向的修订版本是 `c7cd3`，该修订也是目前签出的修订

Git 系统会移动 `master` 指针到新的节点，创建节点之后的情形如图 3-2 所示。在图中我们可以看到，新的提交节点用加粗的红框标记了出来，`master` 分支旧的节点用半透明的形式表示。需要注意的是，`HEAD` 指针并没有改变，它一直是指向 `master` 分支的：

新的提交 `a3b79` 是用加粗的红框标记的，`master` 分支上指针由提交 `c7cd3` 指向了提交 `a3b79` 代表分支发生了变更，如图 3-2 中的虚线所示。



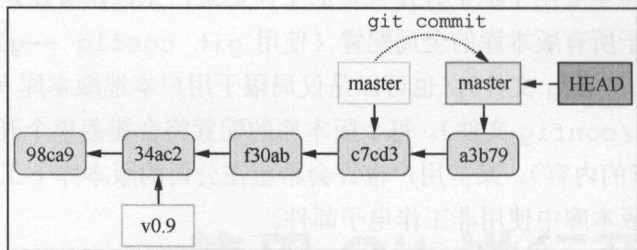


图 3-2 示例项目新增一个提交之后的修订视图 (DAG)，起始状态可以参考图 3-1

### 3.1.2 索引——提交的暂存区

Git 版本库对应的工作区中的每个文件对于 Git 系统来说包括两种，即已知的（跟踪文件）和未知的。其中未知文件对于 Git 系统来说又分为未跟踪和已忽略两类（已忽略文件的详情可以参考第 4 章）。

Git 系统跟踪的文件一般有两种状态：已提交（未变化）和已修改。已提交状态意味着工作目录下的文件内容和最近一次提交的修订内容一致，很安全地存放在版本库中。

如果文件和最新提交的修订版本存在差异，则被认为是已修改的文件。

不过在 Git 系统内部，还存在另外一种状态。接下来让我看看当使用 `git add` 命令添加一个文件后会发生什么。版本控制系统都需要在某处存放上述状态信息。Git 系统采用被称为索引 (index) 的机制实现此功能，它是存储将要提交信息的暂存区。`git add <file>` 命令会暂存文件（当前版本）的当前内容，然后将之添加到索引中。



如果用户只是喜欢将某个文件标记为已添加，那么可以使用 `git add -N <file>` 命令，这样一来上述文件在暂存区的内容就是空白的。

索引是项目的第三个存储拷贝，之前的拷贝一个是包含用户自己项目文件拷贝的工作目录，另外一个为用户本地版本库（存放项目历史记录，专门为用户同步其他开发人员的变更）。

图 3-3 中的箭头表示了 Git 命令拷贝内容的主要步骤，例如 `git add` 命令会将工作区的文件内容拷贝到暂存区。

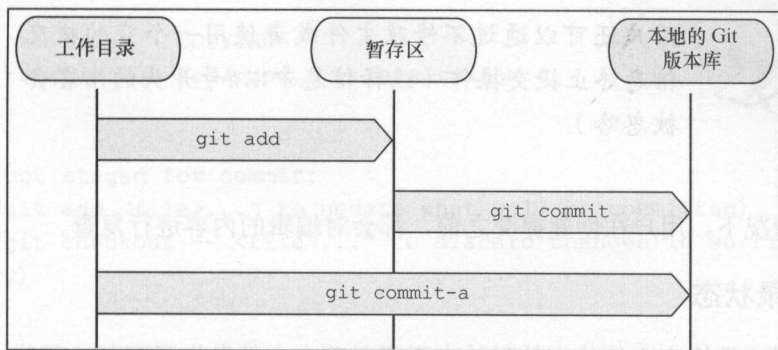


图 3-3 工作目录、暂存区和本地的 git 版本库，  
新增提交的大致流程

创建一个新的提交需要执行如下步骤：

- (1) 在工作目录下面使用编辑器对文件进行编辑。
- (2) 使用 `git add` 命令对上述文件进行暂存，为它们添加快照（文件当前的状态）。
- (3) 使用 `git commit` 命令创建一个新的修订，这会把存放在暂存区的文件信息作为修订版本永久地存放到本地版本库中。

在项目启动之初，工作区的被跟踪文件、暂存区的文件和最近一次提交的修订版本在内容上都是一致的。

不过用户通常会采用提交记录命令的快捷方式，即 `git commit -a`（是 `git commit --all` 的快捷方式），该命令会把被跟踪并且发生变更的文件添加到暂存区中（在当前的 Git 系统中和 `git add -u` 命令的效果一样），然后创建一个新的提交（如图 3-3 所示）。需要注意的是，新增的文件仍然需要使用 `git add` 命令告知 Git 系统对其进行跟踪，然后才能将之添加到新的提交对象中。

### 3.1.3 查看已提交的变更

在提交变更和创建新的修订（新的提交）之前，用户也许希望看看自己的工作成果。

除非用户在命令行中声明了提交注释信息，例如 `git commit -m "简短的描述信息"`，否则 Git 会在提交信息模版中显示将要提交的记录，并且该模版可以根据用户需要进行编辑配置（模版的具体配置可以参考第 10 章）。



用户还可以通过不修改文件或者使用一个空的提交信息终止提交操作（注释信息中以#号开头的内容会被忽略）。

大部分情况下，用户在创建提交之前，都会对编辑的内容进行复查。

## 工作目录状态

用户使用工具检查文件状态的目的在于查看哪个文件发生了变更、哪些地方新增了文件等，这就是 `git status` 命令的主要用途。

上述命令默认输出结果的信息非常详尽。如果项目没有发生变更，例如直接克隆版本库之后，用户会看到类似以下内容的信息：

```
$ git status
On branch master
nothing to commit, working directory clean
```

如果上述分支（本示例中当前用户处于 `master` 分支）是一个本地分支，打算添加一些变更之后发布到公共版本库中，并且事先配置好了对应的上游分支 `origin/master`，那么用户还会看到对应的远程跟踪分支的相关信息：

```
Your branch is up-to-date with 'origin/master'.
```

接下来的示例中，我们将会忽略上述信息。

例如用户打算给项目新增两个文件，一个是包含版权信息的文件，名字叫 `COPYING`；另外一个空白文件，名字叫 `NEWS`。为了跟踪刚才引入的 `COPYING` 文件，需要执行 `git add COPYING` 命令。用户一不留神，使用 `git rm README` 命令将 `README` 文件从工作区删除了；然后编辑了 `Makefile` 文件，用 `git mv` 命令把文件 `rand.c` 的名字改成 `random.c`（并没有修改其中的内容）。

一般来说，完整信息输出格式的好处是易读并且描述信息详尽：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
new file:   COPYING
renamed:    src/rand.c -> src/random.c
```

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

```
modified:   Makefile
deleted:    README
```

Untracked files:

(use "git add <file>..." to include in what will be committed)

```
NEWS
```

如你所见，Git 不仅记录和描述了哪些文件发生了变更，而且还在提交中解释了文件的变更状态，以及哪些将要提交的变更被删除了（git status 命令的详细使用方法可以参考第 4 章）。上述结果大致可以分为 3 个部分。

- **拟提交的变更：**这是已经放入暂存区，准备使用 git commit 命令提交的变更（没有 -a 选项）。它是暂存区的文件快照，和最近一次提交（HEAD）是有显著差异的。
- **未暂存的变更：**这些列表是工作区中和暂存区的快照存在差异的文件列表。这些变更将不会使用 git commit 命令提交，不过可以通过 git commit -a 命令将上述内容作为被跟踪文件的变更提交。
- **未跟踪的文件：**这类文件对于 Git 系统来说是未知的，而且也是可以被忽略的（忽略文件的具体内容可以参考第 4 章）。这类文件可以使用添加操作命令 git add. 在顶层目录中进行批量添加。用户还可以使用 --untracked-files=no（简写为 -uno）选项忽略该操作。

如果用户不希望充分利用暂存区提供的灵活性，那么可以简单地使用 git add 命令，只添加新文件，然后使用 git add -a 命令获取所有被跟踪文件的变更，继而创建提交。这种情况下，用户创建的提交包含将要提交的变更以及不在暂存区的变更。

还有一种简洁的输出格式使用 --short 选项声明。--porcelain 选项适合脚本执行，



因为在保证一致稳定性的同时，使用--short 选项还支持用户对输出结果的编辑。对于某些相同的变更，输出结果和下列内容类似：

```
$ git status -short
A COPYING
M Makefile
D README
R src/rand.c -> src/random.c
?? NEWS
```

采用格式之后，每个路径的状态会用双字母状态码表示。第一个字母表示索引的状态（暂存区和最近一次提交的差异），第二个字母表示工作区的状态（工作区和暂存区的差异）。

并不是所有的状态码组合都是存在的，状态码 A、R 和 C 只可能出现在第一个列，即表示索引的状态。

一个特别的情况是??符号，它主要用来表示未知（未跟踪）文件的，!!符号表示已忽略文件（当使用 git status --short --ignored 命令时）。

值得一提的是，这里并没有列举出所有可能出现的输出信息，例如前文中合并提交导致的合并冲突并没有出现在表 3-1 中，相关内容会在第 7 章详细介绍。

表 3-1 符号及含义

符 号	含 义
	未更新/无变化
M	已修改（更新）
A	已添加
D	已删除
R	已重命名
C	已拷贝

最新修订的差异比较

如果用户不仅希望知道哪些文件发生了变更（使用 git status 命令），而且还想知道具体发生了哪些变更，那么可以使用 git diff 命令。

在上一章节中，我们已经介绍过，Git 系统中包含 3 个区域：工作目录、暂存区和版本

库（通常是最近一次修订版本）。因此，我们不止有一组差异，而是有 3 组，如图 3-4 所示。用户可以要求 Git 回答如下问题：

- 用户编辑了哪些内容但是还未暂存的？换句话说就是，暂存区和工作区之间的差异是什么？
- 哪些内容是已暂存准备提交的？也就是说，最近一次提交（HEAD）和暂存区的差异是什么？

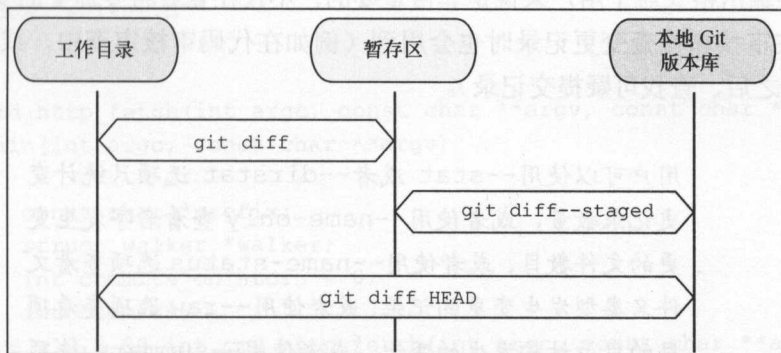


图 3-4 查看工作目录、暂存区和本地 Git 版本库直接的差异

用户希望了解自己编辑了哪些文件但是还未将其暂存，那么可以使用不带任何参数的 `git diff` 命令。该命令会比较用户的工作目录和暂存区直接的差异。上述变更是可以被添加的，但是如果使用 `git commit` 命令提交变更却不会被显示：这些变更并未放入暂存区，因此执行 `git status` 命令之后，查询结果不会包含上述变更。

用户如果希望查看已经暂存的变更并且打算提交这些变更，那么可以使用 `git diff --staged`（或者 `git diff -cached`）命令，该命令会比较最近的一次提交和暂存区之间的差异。上述变更可以使用 `git commit` 命令（不带 `-a` 选项）进行添加，同时也是执行 `git status` 命令后，输出结果中将要提交的变更记录。还可以使用 `git diff --staged <提交记录>` 命令比较暂存区和任意提交历史记录之间的差异；HEAD（最近的一次提交）是默认的比较对象。

可以使用 `git diff HEAD` 命令比较用户当前的工作目录和最近一次提交的修订之间的差异（可以使用 `git diff <提交记录>` 与任意历史修订记录比较差异）。上述变更可以使用 `git commit -a` 命令快速地添加到版本库中。如果用户没有使用 `git commit -a` 命令，而且也不需要充分利用暂存区，那么通常使用 `git diff` 命令查看将要提交的变更记录就可以满足需要了。

如果只使用命令 `git add`，唯一需要处理的问题是新增文件，除非用户使用 `git add --intent-to-add` 命令新增文件到版本库（也可以使用 `git add -N`），否则使用 `git diff` 命令之后，新增的文件将不会显示在查询结果中。

## Git 的统一 diff 格式

一般来说，Git 系统在大部分情况下都会采用统一的 **diff 输出格式** 显示变更记录。

理解这种输出格式对于用户来说是非常重要的，不仅在查看将要提交的变更记录时会用到，而且在审核和检查变更记录时也会用到（例如在代码审核审查中，或者执行 `git bisect` 命令之后，查找可疑提交记录）。



用户可以使用 `--stat` 或者 `--dirstat` 选项只统计变更记录数量，或者使用 `--name-only` 查看名字发生变更的文件数目，或者使用 `--name-status` 选项查看文件名类型发生变更的记录，或者使用 `--raw` 选项查看项目的目录结构发生的变化，或者使用 `--summary` 选项查看扩展首部信息的摘要（后续章节会详细解释扩展首部的含义）。用户还可以使用 `--word-diff` 选项进行字符之间的差异比较，这比行间差异比较精确度更高，这样一来，即使文件内部的标题和段落标题相似，但是只改变段落的格式也可以检测出其中的差异。Diff 生成工具还可以通过设置特定的 **gitattributes** 信息，实现特定文件或者某类文件的差异比较。用户甚至可以指定 **diff 助手**，即描述变更的命令，或者还可以为二进制文件指定文本转换过滤器（详情可以参考第 4 章）。

如果用户喜欢使用图形化工具（通常支持逐行比较）查看上述变更记录，那么可以使用 `git` 的 `difftool` 代替使用 `git diff` 命令。这可能需要预先做适当的配置，详情可以参考第 10 章。

接下来介绍一个使用 `diff` 命令查看 `git` 项目历史记录的高级示例。首先使用 `diff` 命令查看 `git.git` 版本库中的提交 `1088261f`。当然用户也可以使用浏览器查看该提交，例如通过 **GitHub**。下列内容是该提交的第三条补丁记录：

```

diff --git a/builtin-http-fetch.c b/http-fetch.c
similarity index 95%
rename from builtin-http-fetch.c
rename to http-fetch.c
index f3e63d7..e8f44ba 100644
--- a/builtin-http-fetch.c
+++ b/http-fetch.c
@@ -1,8 +1,9 @@
#include "cache.h"
#include "walker.h"

-int cmd_http_fetch(int argc, const char **argv, const char *prefix)
+int main(int argc, const char **argv)
{
+    const char *prefix;
+    struct walker *walker;
+    int commits_on_stdin = 0;
+    int commits;
@@ -18,6 +19,8 @@ int cmd_http_fetch(int argc, const char **argv,
    int get_verbosely = 0;
    int get_recover = 0;

+    prefix = setup_git_directory();
+
    git_config(git_default_config, NULL);

    while (arg < argc && argv[arg][0] == '-') {

```

让我们对上述信息进行逐行分析。

- 第一行是 `diff --git a/builtin-http-fetch.c b/http-fetch.c`，它与 `git` 首部 `diff` 规范格式 `--git a/file1 b/file2` 类似。a/和 b/对应的文件名除了重命名和拷贝操作之外通常是一样的，即使文件的添加和删除操作也是如此。`--git` 选项的含义是 `diff` 代表系统采用的是 `git diff` 的输出格式。
- 接下来的几行是首部扩展信息。本示例的前 3 行告诉我们文件名字由 `builtin-http-fetch.c` 改为 `http-fetch.c`，而且这两个文件的相似度是 95%（用于检测重命名操作的信息）：



```
similarity index 95%
rename from builtin-http-fetch.c
Rename to http-fetch.c
```



扩展首部包含的描述信息是指那些无法在普通的统一 diff 格式中显示的内容（除了文件被重命名的信息）。此外相似（不相似）指数类似文件变更的可描述指标（类似把不可度量的事物转换为可度量的）。

- 本示例中扩展 diff 首部的最后一行是索引 f3e63d7..e8f44ba 100644，它向用户指明了给定文件的模式（100644 表示它是一个普通文件而非一个符号链接，而且它没有可执行权限位；这 3 个文件只有被 Git 系统跟踪的权限）、预映射（给定变更之前的文件版本）和提交映射（给定变更之后的文件版本）的哈希码简写。如果补丁自身无法被顺利合并，那么该行可以通过 `git am --3` 命令尝试执行 3 路合并操作。对于新增的文件，它们的预映射的哈希码是 0000000，对于已删除文件的提交映射哈希码也是如此。

- 接下来是统一的 diff 首部，它包含如下信息：

```
--- a/builtin-http-fetch.c
+++ b/http-fetch.c
```

- 和 `diff -U` 的执行结果相比，它没有源文件（预映射）和目标文件（提交映射）名称被修改时的时间标记。如果新增了文件，那么源文件将会是 `/dev/null`；有文件被删除了之后，那么目标文件也将会是 `/dev/null`。



如果用户设置了 `diff.mnemonicPrefix` 配置变量的值为 `true`，那么首部中的预映射前缀 `a/` 和提交映射前缀 `b/` 是可以替换的，用户可以分别使用 `c/` 前缀表示提交记录，`i/` 前缀代表索引，`w/` 前缀代表工作区，`o/` 前缀代表对象，方便用户进行差异比较。

- 接下来的内容是大块差异比较结果，每个部分显示了若干文件的差异。统一的块状差异格式是以文件内差异所在位置的描述信息为起点的：

```
@@ -1,8 +1,9 @@
```

上述内容采用的格式是@@ 源文件区间 目标文件区间@@。源文件区间的格式是-<起点>, <行数>; 目标文件区间的格式是 +<起点>, <行数>。起点和行数分别代表预映射和提交映射区域的位置和长度。如果没有显示行数, 那么表示行数为0。本示例中, 预映射(文件变更之前)和提交映射(文件变更后)中的变更的起点都是从文件的第一行开始的, 预映射中存在差异的代码片段有8行, 提交映射有9行(有一行是新增的)。一般来说, Git 系统还会显示变更区域周围没有发生变更的3行内容(3行上下文内容)。Git 还会显示相关函数代码发生变更的具体位置(同样的, 用户可以根据需要对其他类似的文件在.gitattributes 属性中进行相关配置), 它和 GNU diff 命令中的-p 选项功能类似:

```
@@ -18,6 +19,8 @@ int cmd_http_fetch(int argc, const char
```

- 接下来是文件的差异描述, 其中包括发生变更的内容和位置。两个文件相同的部分通常使用一个空格(" ")标记开头。两个文件有差异的部分通常在左起打印列是使用如下标记开头的:
  - +: 第二个文件中新增了一行。
  - -: 第一个文件中被删除了一行。



注意, 发生变更的行被着重标记出来的原因是, 该行删除了旧的版本并且添加了新的版本。

在纯文字差异比较格式中, 没有采用逐行比较文件内容的方式, 取而代之的是新增的文字用{+和+}包含, 删除的文字用[-和-]符号包含。

- 如果还包含最后一个区域相关的上下文行, 以及文件内容的最后一行, 通常最后一行是不完整的(这意味着最后一行和文件内部实际的最后一行内容并不是一致的), 你会发现如下内容:

```
\ No newline at end of file
```

本示例中这种情况并未出现。

因此,本书采用的示例中,第一块变更区域的含义是指 `cmd_http_fetch` 被替换成了 `main` 和 `const char *prefix`, 并且新增了一行代码:

```
#include "cache.h"
#include "walker.h"

-int cmd_http_fetch(int argc, const char **argv, const char *prefix)
+int main(int argc, const char **argv)
{
+    const char *prefix;
    struct walker *walker;
    int commits_on_stdin = 0;
    int commits;
```

从上述内容可以看到被替换行的状态变更, 旧版本被删除了(对应的行首用-表示), 新版的内容被添加了(对应的行首用+表示)。换句话说, 在发生变更之前, 文件 `builtin-http-fetch.c` 中相关代码段的部分内容如下:

```
#include "cache.h"
#include "walker.h"

int cmd_http_fetch(int argc, const char **argv, const char *prefix)
{
    struct walker *walker;
    int commits_on_stdin = 0;
    int commits;
```

发生变更之后, 对应的文件被重命名为 `http-fetch.c`, 文件内发生变更的内容如下:

```
#include "cache.h"
#include "walker.h"

int main(int argc, const char **argv)
{
    const char *prefix;
    struct walker *walker;
    int commits_on_stdin = 0;
    int commits;
```

### 3.1.4 可查询的提交

有时,在看过一些未提交的变更记录之后,你也许会发现工作目录下面有两个(或者更多)无甚关联的变更分别属于不同的业务逻辑,它是引起工作拷贝混淆的诱因。用户需要将这些不相关的变更分别放到对应的提交中,即隔离变更集。这种做法也和软件开发的最佳实践不谋而合。一种做法是先创建提交,然后再修复它(将之一分为二),具体的细节可以参考第8章。

不过有时候,其中某些变更急需马上上线(例如一个在线网站的 bug 修复),同时其余变更还有待进一步完善。因此这时用户需要把上述变更分割成两个独立的提交。

#### 文件提交查询

最简单的情形就是这些不相关的变更分布于若干文件中。例如说,如果 bug 只存在于文件 `view/entry.tmpl` 中,并且该文件不存在其他变更,那么可以专门为该文件创建一个修复 bug 的提交,相关命令如下:

```
$ git commit view/entry.tmpl
```

该命令会忽略已经暂存到索引中的变更(即暂存区的内容),取而代之的是提交当前给定文件或目录(工作目录中的变更)中的内容。

#### 变更的交互式查询

不过有时变更无法以这种方式分类,文件中的变更都集中到了一起。用户可以尝试 `git commit` 命令和 `--interactive` 选项一起使用,对上述变更进行梳理:

```
$ git commit --interactive
      staged      unstaged path
1:   unchanged    +3/-2 Makefile
2:   unchanged    +64/-1 src/rand.c
```

\*\*\* Commands\*\*\*

1: status	2: update	3: revert	4: add untracked
5: patch	6: diff	7: quit	8: help

What now>

这里 Git 系统为用户显示了工作区的状态和变更摘要信息,以及暂存区/索引(暂存的)



-状态子命令的输出结果。描述变更的方式是添加和删除文件的数量（和 `git diff --numstat` 输出结果类似）：

```
What now> h
status      - show paths with changes
update      - add working tree state to the staged set of changes
revert      - revert staged set of changes back to the HEAD version
patch       - pick hunks and update selectively
diff        - view diff between HEAD and index
add untracked - add contents of untracked files to the staged set of
changes
*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
```

为了对这些变更进行梳理分类，用户需要用到 `patch` 子命令（例如使用 5 或者 s）。之后 Git 系统会弹出一个 `Update>>` 对话框让用户选择如何处理这些文件，然后用户需要根据上面的状态信息，输入希望更新的文件对应的数字标记，然后按下回车键即可。用户还可以输入 \* 选择所有文件。确认输入完毕目标文件的信息之后，可以按下回车键输入一个空行，告诉系统选择结束了（用户还可以使用 `--patch` 选项直接忽略批量文件的选择）。Git 系统将会为用户逐个显示特定文件的变更区域，然后让用户选择分类，下面的选项是专门操作单个变更区域的：

```
y - stage this hunk (暂存该区域)
n - do not stage this hunk (不暂存该区域)
q - quit; do not stage this hunk or any of the remaining ones (退出，不暂存
任何区域)
```

```
s - split the current hunk into smaller hunks (分隔该区域为更小的区域)
e - manually edit the current hunk (手工编辑当前区域)
? - print help (打印帮助)
```

区域的输出结果和对话提示和下列内容类似：

```
@@ -16,7 +15,6 @@ int main(int argc, char *argv[])
```

```
+
    int max = atoi(argv[1]);
    srand(time(NULL));
    int result = random_int(max);
```

```
printf("%d\n", result);
```

```
Stage this hunk [y,n,q,a,d,/,j,J,g,e,?]? y
```

一般来说,上述选项可以应付大部分在提交内部进行区域选择的操作。不过在一些极端情况下,用户可以将上述区域分割成更小的区域,甚至手工编辑其中的差异。

## 提交创建入门

使用 `git commit --interactive` 进行交互式提交记录查询有一个缺点,那就是它无法对将要提交的变更进行测试。当然用户也可以在创建一个提交之后随时对其进行检查(编译或者运行测试),发现 bug 之后,及时修复它。不过这只是一种替代性的解决方案。用户可以使用 `git add --interactive` 命令将准备提交的变更放在暂存区中,或者采用类似的解决方案(使用 Git 的图形化提交工具,例如 `git gui`)。交互式的提交只是交互式添加变更然后提交的一种快捷方式。之后用户还可以使用 `git diff --cached` 命令查看这些提交,以及使用 `git add <file>`、`git checkout <file>` 和 `git reset <file>` 命令对这些提交做相应的修改。

理论上来说,不管这些变更正确与否,用户都可以对它们进行测试,至少可以确认它们是否可以通过程序编译。为此,首先要使用 `git stash save --keep-index` 命令保存当前的状态,然后将工作目录的状态存放到暂存区中(索引)。执行该命令之后,用户就可以执行测试程序了(至少确认一下程序是否能够通过编译)。如果测试通过,那么用户就可以执行 `git commit` 命令创建一个新的修订版本,如果测试不通过,用户就可以使用 `git stash pop --index` 命令,从暂存区读取工作目录的状态,将之恢复到之前的状态;也有可能需要用到 `git reset --hard` 命令对工作区进行重置。后者可能是非常有必要的,因为 Git 在保存用户的工作记录时过于保守,而且并不知道用户只是将某些记录隐藏暂存了。首先,索引中未提交的变更不能阻止 Git 用其他变更将其覆盖。其次,工作区的变更和暂存区的变更大体相同,因此当然会发生冲突。关于暂存的详细应用,可以参考第 4 章。

### 3.1.5 修改提交

Git 系统非常明显的一个优点是,用户可以随心所欲地恢复其中的任何东西。无论用户如何认真地编辑准备提交的变更记录,或多或少都会出现一些问题,例如忘记添加某个变更或者提交的信息有错别字等。这就是 `git commit` 命令的 `--amend` 选项大显身手的时候。

候了，它能够帮助用户方便地修改最近的提交记录（如图 3-5 所示）。注意，用户还可以使用该选项修改合并提交（例如修复一个合并错误）。



如果用户希望对提交的历史记录做进一步的修改（假定该提交未发布，至少没有人在该提交的基础上提交新的修订），那么可能会用到交互式变基操作或者是某些特殊的工具，例如 StGit（一个基于 Git 的提交历史批量管理工具）。详情可以参考第 8 章的内容。

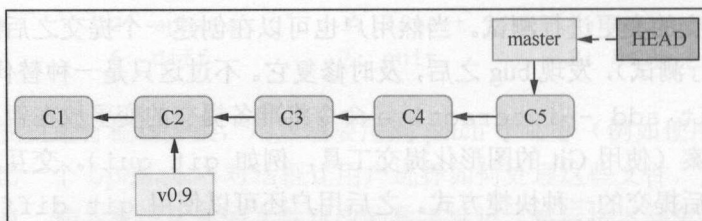


图 3-5 修订的 DAG 图，C1 到 C2 的变更代表对最近一次提交修改之前的状态，C5 代表当前签出的提交。

这里我们使用数字代替 SHA-1 码表示相关的提交记录

如果用户只是想修正提交备注信息中的错误，那么只需要再添加一条备注信息即可，不需要将它暂存（注意，我们使用 `git commit` 命令时并没有使用 `-a / --all` 选项）：

```
$ git commit --amend
```

如果用户希望为最近一次的提交添加一些变更记录（见图 3-6），那么可以先使用 `git add` 命令将这些变更暂存，然后像前面的示例一样再次提交该修订记录，或者使用 `git commit -a --amend` 命令提交这些变更：

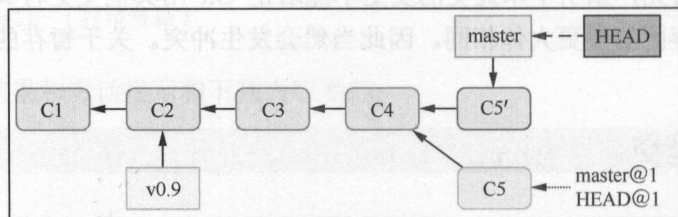


图 3-6 经过编辑的最近一次修订记录（图 3-5）的 DAG 图，这里新的 C5 修订是基于添加了更多变更记录之后的 C5，它替换了旧版本的提交对象


这里有一个非常重要的忠告：你永远不要修改一个已经发布了的修订！这是因为修改操作会创建一个新的提交对象替换原有的对象，如图 3-6 所示。如果开发工作只有一个人的话，那么这么做不会出什么问题。不过如果是多人协作开发，当你把原有的修订记录发布到远程版本库中后，团队其他人员可能已经基于该修订版本做了一些开发工作了。使用经过修订的版本替换原有的版本之后，可能会导致下游出现问题。详细信息可以参考第 8 章。

如果你尝试将一个已经发布过的提交修订，然后将之推送（发布）到某个分支上，Git 将会阻止用户重写已发布的历史提交记录，然后询问用户是否真的希望替换旧的版本进行强制推送（除非用户配置了默认强制推送选项）。修订的历史版本在被编辑之前在分支的引用日志和 HEAD 引用日志中仍然是有效的，例如刚经过修订后不久，它仍然可以通过 `@{1}` 的形式访问。一般来说，除非手动清除，Git 将会保存旧版本修订一个月。

## 3.2 使用分支

分支是一系列开发工作的符号名称。在 Git 中，每个分支都可以看作修订 DAG 中指向某些提交的具名指针，因此它也被称为分支首部。

### 分支在 Git 中的表现形式



目前 Git 在硬盘中采用了两种截然不同的方式来表示分支：松散格式和压缩格式。例如 master 分支（该分支是 Git 采用的默认分支名，用户在创建新的版本库时默认的分支名就是它）。采用松散格式时，它是 `.git/refs/heads/master` 中的一行文本，其中指代分支的内容是十六进制的 SHA-1 码。在压缩格式中，它是 `.git/packed-refs` 中的一行文本，使用最顶部修订的 SHA-1 码和分支全名一起表示该分支。

一系列的开发工作就是指从分支首部为起点所有可达的修订集合。而且它并不一定是线性的修订，还可以是分支分流或者联合。



### 3.2.1 新建分支

用户可以使用 `git branch` 命令创建一个新分支，例如在当前分支上新建一个名为 `testing` 的分支（参见图 3-7 右上部分），可以执行如下命令：

```
$ git branch testing
```

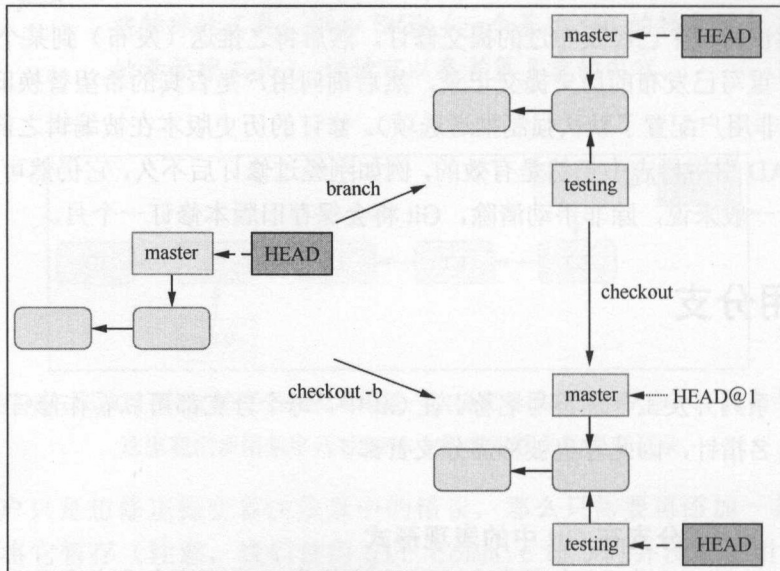


图 3-7 新建一个 `testing` 分支，然后切换到该分支，或者新建一个分支并且马上切换到该分支上（使用命令）

执行上述命令之后会发生什么呢？该命令会为用户创建一个可供访问的新指针（一个新引用）。如果用户希望在创建新分支的同时将它指向某些修订提交，那么可以使用相应的参数。

不过需要注意的是，`git branch` 命令并不会改变 `HEAD` 指针的位置（指向当前分支的符号引用），并且不会改变工作目录中的内容。

如果用户希望创建一个新分支并切换到该分支上（可以马上在新建分支上工作），那么可以使用如下快捷方式：

```
$ git checkout -b testing
```

如果我们在当前的版本库中创建分支，`checkout -b` 命令的差异仅在于它还会将 `HEAD` 指针移动到新建的分支上，如图 3-7 所示。

## 3.2.2 孤儿分支

有时用户也许希望在版本库中新建一个和主分支没什么关联的孤儿分支。例如为每个预览版程序单独存放对应的用户手册，使得用户不需要安装转换工具或者解析器（例如 AsciiDoc 解析器）就可以方便地阅读用户手册（例如 HTML 格式的帮助页面）。又或者用户希望在版本库中的每个项目存放一些 Web 页面，这和 GitHub 网站项目中 Web 页面的用途类似。有些用户希望开源自己的项目代码，但是首先要把代码清理一番（例如修改版权和授权许可文件）。

一种办法是为包含上述内容的孤儿分支建立一个独立的版本库，然后从远程跟踪分支上获取它们，然后用户可以基于该孤儿分支创建一个本地分支。

用户还可以执行如下命令达到上述目的：

```
$ git checkout --orphan gh-pages
Switched to a new branch 'gh-pages'
```

上述命令会重新生成某些状态，在执行完 `git init` 操作之后将 HEAD 指针指向 `gh-pages` 分支，如果该分支不存在的话，会先创建该分支然后执行上述操作。而且该分支的创建是基于第一个提交对象的。

如果用户希望使用类似 GitHub 页面这种简洁的目录结构启动项目，那么还需要将起点分支的内容移除（HEAD 指向默认分支，即当前分支和当前工作目录的状态）。例如使用如下命令：

```
$ git rm -rf .
```

对于开源目代码中需要排除的私人信息，用户需要在工作目录中认真编辑，进行相应的替换。

## 3.2.3 分支的查询和切换

为了切换到一个已有的本地分支上，用户需要执行 `git checkout` 命令。例如在创展 `testing` 分支之后，可以使用如下命令切换到该分支上：

```
$ git checkout testing
```

分支切换的主要过程可以参考图 3-7 中右上部到右下部的状态迁移。

## 分支切换释疑

在切换分支时，Git 也会将签出的内容放入工作目录中。那么如果有未提交的变更，又会发生什么呢（先不考虑 Git 将会切换到哪个分支上）？



切换分支时，让当前分支保持整洁的状态是一个好习惯。如果有必要的话，在切换前清空暂存区或者创建一个提交。在极个别情况下，将未提交的变更一并执行分支切换操作是很有用的，具体的内容后续章节会进行介绍。

如果当前分支对应的文件变更和将要切换的目标分支没有关联，那么当前分支未提交的变更也会一并移动到新分支中。这对于用户开始着手研究新的东西时非常有用。随着时间的推移，用户正好可以将这一系列的工作放到独立的特性分支中。如果未提交的变更和给定分支有冲突，那么 Git 将会拒绝切换到目标分支，从而防止用户的工作成果遭到破坏：

```
$ git checkout other-branch
error: Your local changes to the following files would be overwritten by
checkout:
    file-with-local-changes
Please, commit your changes or stash them before you can switch branches.
```

在这种情况下，有以下几个解决方案可供选择。

- 用户可以隐藏自己的变更，当切换到原来的分支时再恢复它们（通常这是比较推荐的做法）。或者可以简单地为这些正在进行中的工作创建一个临时提交，然后在切换到原来分支时，可以对该提交进行修改或回退。
- 用户还可以尝试通过合并操作将自己现有的变更记录移动到新分支中，可以使用 `git branch --merge` 命令（该操作会执行三路合并，其中包括当前分支、工作区中未保存的变更和目标分支）；还可以在签出之前隐藏用户的变更记录，然后在切换分支之后将隐藏的变更记录恢复到暂存区中。
- 用户还可以使用 `git checkout --force` 命令丢弃原有的变更。

## 匿名分支

如果用户尝试签出一个非本地分支会如何呢？例如任意的修订（像 `HEAD^`），或者一

个标签（像 v0.9），或者一个远程跟踪分支（像 origin/master）。在这种情况下，Git 会假定用户需要在当前工作目录状态的基础上创建一个提交。

旧版本的 Git 程序会拒绝执行没有目标分支的切换操作。不过现在的 Git 程序会通过 HEAD 指针分离创建一个匿名分支，即直接指向一个提交对象，而不是使用一个引用符号指向一个分支，如图 3-8 所示。为了在当前位置显式创建一个匿名分支，用户可以在执行 checkout 命令时使用 --detach 选项。被分离的 HEAD 引用会在分支列表中被当作 Git 的一个历史修订版本存在，又或者是作为较新的版本（修订从 HEAD 脱离或者 HEAD 在某处脱离）。

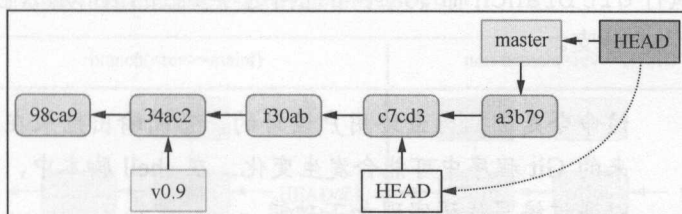


图 3-8 无分支的签出结果，Git 中签出 HEAD 操作之后的状态，HEAD 被分离了，或者称之为匿名分支

如果用户因为失误而将 HEAD 和分支脱离了，那么还可以将之恢复到脱离之前的状态（这里的“-”代表上一分支）：

```
$ git checkout -
```

因为当用户在创建一个匿名分支时，Git 会给用户一些警示信息，因此用户还可以使用 `git checkout -b <new-name>` 命令为该分支命名。

## Git 的智能签出

还有一种非常特殊的情况属于签出时的内容不是分支的。如果你使用分支的简称（本示例是 next）签出了一个远程跟踪分支（例如 origin/next），但是如果当前状态下有一个同名的本地分支的话，Git 会认为用户希望在远程跟踪分支的基础上添加一些新的内容，可以完成用户的预期目标。“做我所想”（DWIM）机制将会创建一个新的本地分支，以便跟踪远程跟踪分支。

相关命令如下：

```
$ git checkout next
```

它等效于如下命令：



```
$ git checkout -b next --track origin/next
```

只有在不会产生歧义的情况下 Git 才会执行上述操作：本地分支必须不存在（否则该命令会简单地切换到目标分支），而且必须只有一个远程跟踪分支与之对应。这种情况可是使用 `git show-ref next`（使用分支简称）命令检查远程跟踪分支是否只有唯一符合条件的记录（最近的一条记录可以使用 `refs/remotes/`前缀和引用名进行识别）。

## 3.2.4 分支列表

如果用户在执行 `git branch` 命令时不带任何选项参数的话，那么它会显示所有的分支，并使用\*标记当前分支。



该命令是专门为最终用户设计的，它的输出结果在未来的 Git 程序中可能会发生变化。在 shell 脚本中，可以通过编写代码实现如下功能。

- 可以使用 `git symbolic-ref HEAD` 命令获取当前分支的名称。
- 可以使用 `git rev-parse HEAD` 命令找到当前提交的 SHA-1 码。
- 可以使用 `git show-ref` 或者 `git for-each-ref` 命令显示所有分支。

上述命令都是管道化的，因此可以将它们应用到脚本中。

用户可以使用 `-v` (`--verbose`) 或者 `-vv` 选项查询更多信息；还可以对分支进行条件查询，例如可以在给定的 shell 程序中使用 `git branch --list <模式>` 命令进行模式匹配查询（如果有必要的话，可以使用引号将模式包裹起来，防止它被 shell 解析）。

用户可以使用 `git remote show` 命令查询远程版本库的信息，当然其中也包括远程分支的查询，详情可以参考第 6 章。

## 3.2.5 分支的回退和复位

如果用户想丢弃最近一次提交的修订记录，或者回退（重置）当前的分支到上一分支，他该怎么做？为此，用户需要使用 `reset` 命令。它可以改变当前分支指针的指向。不过需要注意的是，和 `checkout` 命令不同，`reset` 命令在默认情况下不会改变工作目录的内容，

用户需要使用相应的命令参数才行，例如 `git reset --keep` 命令（尝试保留未提交的变更）或者 `git reset --hard`（强行删除未提交的变更）。

`reset` 命令对工作区的影响会在第 4 章详细介绍。

图 3-9 展示了在指定分支和无分支参数的情况下 `checkout` 命令和 `reset` 命令之间的差异。总之，`reset` 命令会改变当前分支指针的指向（移动引用），而 `checkout` 命令既可以进行分支切换，又可以在没有无分支的情况下在给定的修订中和 `HEAD` 分离。

从图 3-9 可以看出，左上角的修订图形展示了执行 `git checkout maint` 命令之后的结果，并且它的起始状态是图片中心的结构提供的。

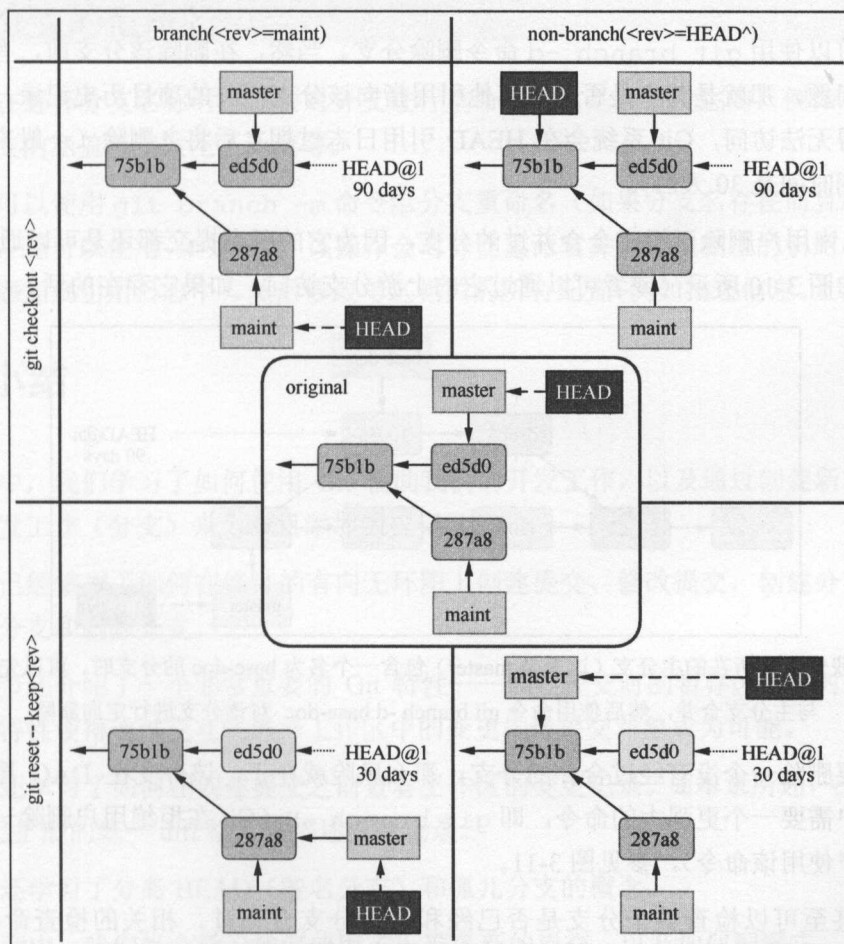


图 3-9 在指定分支（例如 `maint`）和无分支（例如 `HEAD^`）参数的情况下，`checkout` 命令和 `reset` 命令之间的差异对照

### 3.2.6 分支的删除

在 Git 中，一个分支其实就是一个指针，它在修订的 DAG 图上就是一个指向节点的外部引用，因此删除一个分支实际上就是删除一个指针。



实际上当删除一个分支时，同时也永久性地（至少对于当前版本的 Git 是如此）移除了和该分支对应的引用日志，即它的本地历史日志。

用户可以使用 `git branch -d` 命令删除分支。当然，在删除该分支前，有一个必须考虑的问题，那就是其中是否还有其他引用指向该分支相关的项目历史记录。否则该修订将会变得无法访问，Git 系统会在 HEAD 引用日志过期之后将之删除（一般来说，默认配置的过期时间是 30 天后）。

Git 允许用户删除已经完全合并过的分支，因为它的所有提交都还是可以通过 HEAD 访问的，如图 3-10 所示（或者可以通过它的上游分支访问，如果它存在的话）。

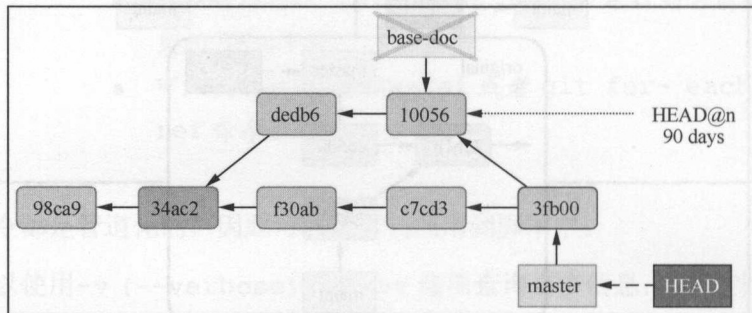


图 3-10 当我们目前所在的主分支（这里是 master）包含一个名为 base-doc 的分支时，可以先将它的变更与主分支合并，然后使用命令 `git branch -d base-doc` 对该分支进行定向删除

如果要删除一个没有经过合并的分支，那么风险就在于，该分支在 DAG 图上是不可达的。用户需要一个更强大的命令，即 `git branch -D`（Git 在拒绝用户删除一个分支时会建议用户使用该命令），参见图 3-11。

用户甚至可以检查某个分支是否已经和其他分支合并过，相关的检查命令是 `git branch --contains <branch>`。

用户不能删除当前的分支。

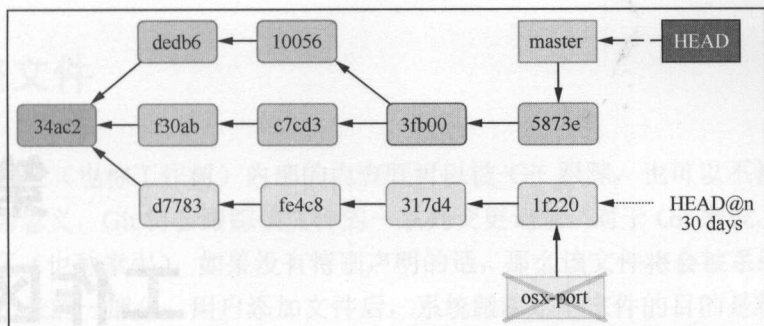


图 3-11 使用 `git branch -D osx-port` 命令删除一个未经过合并的分支 `osx-port`

## 3.2.7 分支的重命名

有时在选择分支名称时常需要对分支进行重命名。这是经常发生的，例如在开发过程中特征分支的涵盖范围发生了改变等。

用户可以使用 `git branch -m` 命令给分支重命名（如果分支名存在而且用户希望重写该分支，还可以使用 `-M` 选项）；该操作会对分支重命名并且移动相应的引用日志（将重命名操作添加到引用日志中），然后修改与之相关的所有配置（例如描述信息、上游信息等）。

## 3.3 小结

本章中，我们学习了如何使用 Git 辅助我们的开发工作，以及通过创建新的提交和一系列的开发工作（分支）来为项目添加历史记录。

我们已经掌握了如何在修订的有向无环图上创建提交、修改提交，创建分支、切换分支、回退分支和删除分支。

本章节还介绍了一个非常重要的 Git 特性——创建提交时的暂存区，有时也称之为索引。这一特性使得通过交互式选择工作区中的变更构造提交对象成为可能。

我们还学习了如何在创建提交之前查看工作区的变更记录。如本章所述，在细节方面，Git 采用已扩展的统一 diff 格式来描述变更记录。

我们还学习了分离 HEAD（匿名分支）和孤儿分支的概念。

下一章中，我们将会学习如何使用 Git 准备新的提交，以及如何配置它，从而使得我们的工作更简单一些。我们还会学习对工作区、暂存区和项目历史中的内容的查看、搜索和应用。同时还会学到如何使用 Git 处理程序中断和故障恢复等问题。



## 第 4 章

# 工作区管理

上一章已经介绍过如何使用 Git 进行程序开发，以及如何创建新的修订。现在我们会把重点放在学习如何管理工作目录，以便用户可以更高效地新建提交。

同时本章还会向读者详细介绍如何管理文件。为了满足用户处理一些特殊文件管理的需求，本章还引入了忽略文件和文件属性等概念。

读者还会学习如何在处理工作区和暂存区文件的过程中修复错误，以及如何修复最近一次提交记录中的问题；同时还会了解到如何安全地使用暂存机制和多个工作目录处理工作流中的中断问题。

上一章中读者已经学习了如果查看变更；接下来的章节会介绍如何有选择地撤销和重做变更记录，以及如何查看文件不同版本之间的差异。

本章的主要内容有以下几点。

- 忽略文件：特意不让版本控制器管理某些文件。
- 文件属性：特定路径的配置。
- 处理文件和二进制文件。
- 为了提高版本库的可移植性，提供对文件尾行转换的支持。
- `git reset` 命令在不同模式下的应用。
- 为了处理中断而隐藏暂存变更。
- 搜索和查看文件。
- 交互式文件重置和恢复文件变更。
- 通过移除未跟踪文件来保持工作区整洁。

## 4.1 忽略文件

用户的工作区（也称工作树）内部的内容既可以被 Git 跟踪，也可以不被它跟踪。被跟踪文件，顾名思义，Git 将会跟踪该文件的一系列变更记录。对于 Git 来说，如果一个文件存在于暂存区（也称索引），如果没有特别声明的话，那么该文件将会被系统跟踪，并作为下一个修订记录的一部分。用户添加文件后，系统跟踪这些文件的目的是将它们当作项目历史的一部分。



索引或者暂存区不仅可以用来告知 Git 系统跟踪哪些文件，而且可以作为新建提交的一种暂存器，具体信息可以参考第 3 章。而且索引或者暂存区可以帮助用户解决合并冲突，详情可以参考第 7 章。

通常对于某些独立文件或者某类文件来说，用户永远都不希望它们作为项目历史记录的一部分而存在，并且也不希望系统跟踪它们。这些文件可能是编辑器的备份文件，也有可能是项目编译系统自动生成的临时文件。

用户不希望 Git 系统自动添加上述文件，例如，在使用 `git add :/`（添加整个工作区）和 `git add .`（添加当前目录下的所有文件）命令批量添加文件时，或者使用 `git add --all` 命令更新工作区状态索引时。

另外一方面，用户又希望 Git 能够有效地防止将不必要的文件添加到系统中。同时用户还希望在执行 `git status` 命令后不显示这些文件，因为它们的数量可能会非常庞大。有时它们也可能会和一些新增的未知文件混在一起。因此用户希望这类文件特意不被跟踪，即忽略它们。



### 未跟踪和重新跟踪的文件

如果用户希望忽略某个以前被跟踪的文件，例如从手工生成 HTML 文件迁移到使用类似 Markdown 这样的轻量级标记语言时，用户通常需要在不将它们从工作目录中删除的情况下，将它们添加到忽略文件列表中，对它们取消跟踪。为此，用户可以使用 `git rm--cached <文件名>` 命令。为了添加（开始跟踪）一个特意不跟踪（即忽略）的文件，用户需要使用命令 `git add -f`。

### 4.1.1 将文件刻意标记为不跟踪的

这种情况下，用户可以通过 Git 系统中 `gitignore` 文件添加一组 shell glob 模式来指定希望忽略的文件，文件中每个模式占用一行的位置。

- 可以通过配置变量 `core.excludesFile` 指定每个用户的个性化配置文件，该变量默认的值是 `$XDG_CONFIG_HOME/git/ignore`。如果环境变量 `$XDG_CONFIG_HOME` 未设置或者为空的话，那么其默认值为 `$HOME/.config/git/ignore`。
- 每个本地版本库的 `$GIT_DIR/info/exclude` 文件在本地版本库克隆的管理区中。
- `.gitignore` 文件在项目工作区目录下；该文件通常是被系统跟踪记录并且可以和其他开发人员共享的。

一些诸如 `git clean` 的命令还支持用户通过命令行声明忽略模式。

在判断是否忽略某个路径时，Git 系统会根据上述列表中的模式以一定的顺序进行模式匹配，然后根据就近优先原则决定输出结果。`.gitignore` 文件也是按照一定的顺序进行检查的，它会从项目的顶级目录开始，依次遍历项目中的所有文件。

为了增强 `gitignore` 文件的可读性，用户可以使用空白行将文件分组（空白行不匹配文件）。用户还可以对模式进行描述或者使用附带注释的模式组，以 `#` 开头的代表一行注释（为了实现对一个 `#` 开头的哈希字符串进行模式匹配，通常会在第一个哈希字符前面使用 `\` 对它转义，例如 `\#*#`）。字符尾部的空格会被忽略，除非使用 `\` 对它进行转义。

`gitignore` 文件中的每一行都代表一种 UNIX 的 glob 模式，即 shell 通配符。通配符 `*` 可以匹配 0 个或者多个字符（任意字符串），通配符 `?` 可以匹配任意单个字符。读者还会了解到字符类方括号 `[...]` 的使用，例如下面的模式匹配示例：

```
*.[oa]
*~
```

这里的第一行内容是告诉 Git 系统忽略所有后缀名是 `.a` 或者 `.o` 的文件（例如静态链接库），以及软件编译过程中产生的临时文件。第二行是告诉 Git 系统忽略所有以 `~` 结尾的文件，这类文件常见于很多 UNIX 文本编辑器采用的临时备份文件。

如果模式中不包含 `/`，即文件路径的分隔符，Git 会将它视为一个 shell glob 通配符，



并且根据它查找相应的文件名和目录名，例如 `.gitignore` 文件的路径或者某个版本库顶级目录。以/结尾的模式是一个例外，它主要是用来匹配目录的，除非目录级下的反斜杠被移除了。以反斜杠开头的模式是用来匹配路径名称前面位置的，它的含义有以下几种。

- 模式中不包含反斜杠匹配版本库中的任意路径，那么我们可以说该模式是递归的。

例如 `*.o` 模式匹配任意的文件对象，其中既包含 `gitignore` 文件，也包括 `file.o` 和 `obj/file.o` 这样的子目录。

- 以一个反斜杠结尾的模式只匹配目录，否则它就是递归的（除非它还包含其他斜杠）。

例如 `auto/` 模式将会匹配顶层的 `auto` 目录以及 `src/auto` 目录，但是它不会匹配名为 `auto` 的文件（或者一个标记链接）。

- 如果希望固定一个模式，并且确保它是非递归的，那么可以在其起始位置添加一个反斜杠进行转义，例如 `/TODO` 文件将会忽略当前层级的 `TODO` 文件，但是不会忽略子目录中的文件，例如 `src/TODO`。
- 包含斜杠的模式都是固化并且非递归的，通配符不会匹配作为目录分隔符的斜杠。如果用户希望匹配任意目录，那么可以使用两个连续的星号 `**` 替换路径地址的某个部分（例如 `**/foo`，`foo/**` 和 `foo/**/bar`）。

例如 `doc/*.html` 匹配的是 `doc/index.html` 文件，但是无法匹配地址 `doc/api/index.html`。为了匹配 `doc` 目录下的任意 HTML 文件，用户可以使用 `doc/**/*.html` 模式（或者将 `*.html` 模式添加到 `doc/.gitignore` 文件中）。

用户还可以在模式前面加一个感叹号！前缀使之失效，任何根据先前的规则被排除的文件，现在又再次被包含（不再被忽略）进来了。例如已经忽略所有生成的 HTML 文件，但是希望包含某个通过手工生成的文件，用户可以在 `gitignore` 文件中做如下设置：

```
# 忽略所有通过 AsciiDoc 源代码工具生成的 HTML 文件
*.html
# 下列有手工生成的文件将会是个例外
!welcome.html
```

注意，Git 基于性能方面的考虑不会排除某个目录（至少 2.7 版的程序是如此），这意味着当父目录被排除后，用户不能将其目录下的某个文件再次包含到跟踪列表中。也就是说，为了将某个子目录作为例外被跟踪，必须做如下配置：



```
#除了目录 t0001/bin 之外，将会排除所有文件。  
/*  
!/t0001  
/t0001/*  
!/t0001/bin
```

为了匹配一个用 ! 开始的模式，必须使用一个反斜杠对其进行转义，例如模式 `\!important!.md` 是为了匹配 `!important!.md`。

## 4.1.2 确定忽略文件类型

现在我们已经学习了如何将文件状态特意标记为不被跟踪的（忽略），那么接下来的问题是哪些（哪一类）文件应该被标记。另外一个问题是：上述文件的位置是什么？以及我们应该如何在 3 个 `gitignore` 文件中声明需要忽略的文件类型？

首先，用户永远不应该跟踪自动生成的文件（通常是由项目的编译系统生成的）。如果用户将它们添加到版本库中了，那么它们很有可能无法和源文件保持同步。此外，它们也不是必须添加的文件，因为系统可以很容易地重新生成它们。唯一的例外是生成这些文件的源文件极少发生变动，并且生成它们需要额外的工具辅助，但是开发人员有可能没有这些工具（如果源代码经常发生变更，用户可以使用一个孤儿分支存放这些生成的文件，只在发布预览版程序时更新该分支）。

这些是所有开发人员都希望忽略的文件，因此它们应该被放到一个被跟踪的 `.gitignore` 文件中。模式列表将会是经由版本控制的，并且可以通过克隆副本分发给其他开发人员。读者可以在 <https://github.com/github/gitignore> 上找到和若干编程语言有关的一组非常有用的 `.gitignore` 模版。

其次，临时文件和特定产品相关的用户工具链，这些内容通常都不会与其他开发人员共享。如果模式对版本库和用户都是有效的，例如在版本库中的辅助文件，并且该文件也会影响特定的工作流用户（例如该项目中 IDE 程序），那么它就应该被放到每个克隆的 `$GIT_DIR/info/exclude` 文件夹中。

在用户采用的通用忽略模式中，不必特意声明版本库（或者项目），一般来说可以通过 `core.excludesFile` 配置变量中进行声明，同时对于每个用户（全局）还可以在 `~/.gitconfig`（或者 `~/.config/git/config`）配置做相关设置。一般来说，默认的配置文件的默认路径是 `~/.config/git/ignore`。



专属于每个用户的忽略文件应该不会是 ~/.gitignore, 因为如果用户希望让 ~/directory (\$HOME) 主目录保持版本控制, 那么该文件有可能就会是用户主目录对应版本库中的 .gitignore 文件。

这里是编辑器或者 IDE 程序生成的备份或者临时文件对应的模式匹配文件所在之处。



已忽略文件通常也是无关紧要的

警告: 不要将重要资料添加到忽略列表中, 这些资料通常是用户不希望在某个版本库中被跟踪的, 但是内容相比忽略文件列表中的内容来说却是非常重要的! 被 Git 忽略的这类文件既可以很容易地被重新生成(产品编译系统生成的中间文件), 而且对于用户来说又是无关紧要的(临时文件或者备份文件)。

因此 Git 会认为这类已忽略的文件用处不大, 当需要做一些清理工作时, 即使在没有向用户提示的情况下也可能把它们删除, 例如, 如果已忽略文件和当前签出的修订内容有冲突时。

### 4.1.3 忽略文件列表

用户可以在执行 status 命令时使用 --ignored 选项查看被忽略的文件:

```
$ git status --ignored
On branch master
Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
```

```
rand.c~
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git status --short --branch --ignored
## master
!! rand.c~
```

用户还可以使用清理被忽略文件的测试选项: git clean -Xnd 和底层(管道化)

的命令 `git ls-files`:

```
$ git ls-files --others --ignored --exclude-standard  
rand.c~
```

后一个命令还可以用来显示匹配忽略模式的被跟踪文件。找到这类文件往往意味着某些文件需要被取消跟踪（也可能是源代码文件临时生成的中间文件），或者也可能是忽略模式覆盖的范围过于宽泛了。因为 Git 是通过暂存区（缓存）已有的文件来识别哪些文件需要被跟踪的，相关的命令如下：

```
$ git ls-files --cached --ignored --exclude-standard
```



### 底层命令（Plumbing）和高层命令（Porcelain）的区别

Git 的命令主要分为两种：一种是方便和用户交互的高层命令，另外一种是为方便编写 shell 脚本的管道化底层命令。它们之间的区别在于：高层命令的输出结果可以变更并不断完善，例如在遇到与 HEAD 分离的情况下，执行 `git branch` 命令后的输出结果显示了其中的分离过程（无分支到与 HEAD 分离）。同时底层命令中也有选项（通常是 `--porcelain`）来决定是否选择无变更的输出结果。它们的输出结果和行为是可以根据主题配置的。

另外一个比较重要的区别是高层命令会尝试猜测用户的意图，并且会使用默认参数和默认配置。底层命令则没有那么智能，用户必须通过 `--exclude-standard` 这样的选项和 `git ls-files` 命令搭配使用，才能让它采用忽略文件的默认设置。详情可以参考第 10 章。

## 4.1.4 忽略跟踪文件内的变更

也许用户版本库中不少文件发生了变更，但是它们很少被提交。这类文件可以是若干本地配置文件，为了适应用户本地环境经过编辑配置，但是用户永远不希望将它们提交到远程上游分支。这也可以是某个包含新发布的预览版软件名称的文件，只有当它们添加了下一个预览版程序便签后才会被提交。用户可能希望让这类文件大部分时间都保持“杂乱

无章”的状态，但是又不希望 Git 系统经常向用户提示这些文件发生了变更。为了防止干扰其他变更信息的提示，用户应该将这类信息忽略。

用户可以对 Git 进行配置，让它跳过对工作目录的检查（假定它始终是最新版本），并使用文件的暂存版本替代，可以对某个文件设定相应的 `skip-worktree` 标记来达到此目的。为此用户将会需要用到底层的 `git update-index` 命令，它相当于面向用户的高层命令 `git add`（用户可以使用 '`git ls-files`' 查看文件的状态和标记）：

```
$ git update-index --skip-worktree GIT-VERSION-NAME
$ git ls-files -v
S GIT-VERSION-NAME
H Makefile
```

不过这种对工作区的省略也会影响到 `git stash` 命令；为了暂存用户的变更并且保持工作目的整洁，用户需要禁用该标记（至少是临时措施）。为了让 Git 再次监测到工作目录中的修订版本，并且开始跟踪文件的变更记录，可以执行下列命令：

```
$ git update-index --no-assume-unchanged GIT-VERSION-NAME
```



还有一个类似的选项 `assume-unchanged`，它可以用来让 Git 系统完全忽略文件的变更，甚至可以假定文件没有发生任何变化。当文件被这个标签标记之后，它们将永远不会出现在 `git status` 或 `git diff` 命令的输出结果中，与之相关的变更将不会被暂存或提交。有时这是非常有用的，特别是在检查一个大型项目的文件变更时。不要对被跟踪的文件使用 `assume-unchanged` 选项已达到忽略文件的目的。用户务必确保文件没有发生变更，不会发生欺骗 Git 系统的情况，例如 `git stash save` 命令将会根据你的设置进行保存，这样就可能失去用户本地文件的变更记录。

## 4.2 文件属性

Git 中有一些配置和选项可以用来声明基本路径，它的机制和忽略文件类似（将文件特意标记为不跟踪的）。这些路径声明设置被称为属性。



为了给文件指定一些属性以便匹配给定的模式，用户需要在下列 `gitattribute` 文件中添加一行由空格分隔的属性集合作为模式（它的机制和 `gitignore` 文件类似）。

- 单个用户配置文件：对于每个用户来说，文件中的属性会影响与该用户有关的所有版本库，该文件默认路径是 `~/.config/git/attributes`，并且是通过配置变量 `core.attributesFile` 声明的。
- 版本库属性文件：该文件一般在本地版本库克隆的管理区中，文件路径是 `.git/info/attributes`，这些属性只会对声明的单个版本库克隆产生影响（对于单个用户工作流程来说）。
- 项目中的 `.gitattributes` 文件，其中的属性可以和团队其他成员共享。

模式匹配文件的规则和前面提到的 `gitignore` 文件类似，除非存在不兼容的省略模式。

对于给定路径，每个属性可以设置为下列几个状态：设置（设定值为 `true`），未设置（设定值为 `false`），以及给定值或者 `unspecified`：

```
pattern* set -unset set-to=value !unspecified
```

注意，用字符串给一个属性赋值时，等号（=）两边是没有空格的！

当存在多个模式匹配路径时，后一行的属性会覆盖前一行的基本属性。`Gitattribute` 文件采用顺序优先原则，即在给定目录中从单个用户文件到 `.gitattributes` 文件，这和 `gitignore` 文件类似。

## 识别二进制文件和尾行转换

根据操作系统和应用程序的不同，表示文本文件中新行的方式各不相同。UNIX 和类 UNIX 系统（包括 Mac OS X）采用一个单独的控制符 LF（`\n`）表示。与此同时，微软的 Windows 系统采用控制符 LF 后紧跟 CR（`\n\r`）表示，Mac OS 系统从第 9 版开始只使用 CR 表示（`\r`）。

如果开发团队成员采用了不同的操作系统，这可能会导致开发便携式应用时出现问题。我们不愿意因为换行符的不同而产生一些无关紧要的变更记录。为此 Git 系统能够在版本库的提交操作（`check-in`）中自动将换行符（`eol`）统一转换成 LF，同时可以在签出状态下的工作目录中将它们转换为 CR+LF 形式。

用户甚至可以通过文本属性决定一个文件是否应该遵循换行符的约定。设置了该属性之后就启用了换行符约定，不设置的话就会禁用该功能。将该值设置为 `auto` 之后，Git 系

统会自动处理文本文件；如果是文本文件，Git 会对它启用换行符约定。如果文件的文本属性未设置，Git 系统会使用 `core.autocrlf` 的设置来决定是否将它们当作 `text=auto` 的情况来处理。



#### Git 如何检查一个文件中是否包含二进制数据

为了判别一个文件中是否包含二进制数据，Git 会检查文件起始部分产生零字节的位置（空字符或者是 `\0`）。当在决定是否对一个文件应用转换规则（即换行约定）时，标准会更严格：如果一个文件是文本构成的，那么它必须不能为空，而且其中非打印字符不得超过内容总量的 1%。

这意味着 Git 通常会把以 UTF-16 编码的文件当作二进制的。

在工作目录下，为了让 Git 决定文本文件该选择哪一种换行约定，用户需要设置 `core.eol` 配置变量。其中的值有 `crlf`、`lf` 和 `native` 可供用户选择（最后一种是默认格式）。用户还可以对给定文件使用 `eol=lf` 和 `eol=crlf` 强制指定换行约定规范。表 4-1 展示了向下兼容的 `crlf` 属性。

表 4-1

向下兼容的 `crlf` 属性

旧的 <code>crlf</code> 属性	新的文本和 <code>eol</code> 属性
<code>crlf</code>	<code>text</code>
<code>-crlf</code>	<code>-text</code>
<code>crlf=input</code>	<code>eol=lf</code>

在极个别情况下，换行操作过程会导致损坏数据。如果用户希望 Git 在处理混合使用 LF 和 CRLF 两种换行符的文件时能够向用户发出警告或者阻止相关的转换操作，那么可以使用 `core.safecrlf` 配置变量进行设置。

有时 Git 也可能无法正确地检测二进制文件，也有可能某些文件就是普通的文件，但是不方便用户识别。例如 PostScript 的脚本文件（\*.ps）和 Xcode 的编译设置文件（\*.pbxproj）。这类文件并不常见，并且比较文本差异的意义不大。用户可以使用二进制属性宏将它们显式标记为二进制的（和 `-text` `-diff` 选项功能相同）：

```
*.ps          binary
*.pbxproj     binary
```



### 启用强制换行转换功能

当版本库中启用了规范化换行功能后(编辑`.gitattributes`文件),用户也应该强制要求对其中的文件遵循此操作,否则因换行操作产生的差异会影响文件的属性变更记录:

```
$ rm .git/index
$ git reset
$ git add -u
$ git add .gitattributes
```

用户可以在执行完`git reset`命令之后查看哪些文件执行了规范化操作,不过检查工作要在执行`git add -u`命令之前。

## 4.2.1 配置 Diff 和 merge

在 Git 中,用户可以通过属性功能配置如何显示文件不同版本之间的差异,以及如何对文件执行三路合并。该特性可以用来扩展上述操作的功能,使得差异比较结果内容更丰富,合并操作出现冲突的概率更小。它甚至可以用来高效地比较二进制文件。为此我们一般需要设置差异比较(diff)和合并操作(merge)的驱动。

属性文件只能告诉用户使用哪些驱动,其余的信息都在配置文件里面,而且配置信息并不是像`.gitattributes`文件那样自动就可以和其他开发人员共享的(用户还可以创建一个共享配置片段,然后将它添加到版本库中,方便和其他开发人员使用相对路径引用它)。方便易用的工具配置信息在不同电脑和操作系统上的表现可能不尽相同,不过这也意味着某些信息对跨平台的要求要高一些。

不过,幸好系统内置了不少差异比较和合并操作的驱动供用户选择。

### 生成 diff 和二进制文件

在对某些特殊的文件进行差异比较时,该文件的 diff 属性会影响比较的结果。如果该属性未设置,那么 Git 会在执行差异比较时将它们当作二进制文件处理,并且只显示二进制文件的差异(二进制数据差异)。设置该选项之后,即使文件中包含标志二进制文件的字



节序列，例如空字符（\0），Git 系统也会强制将文件当作文本来处理。

用户可以通过 `diff` 属性来让 Git 系统高效地显示一个二进制文件不同版本之间的差异。为此，用户有两个选项：比较容易的一个办法是告诉 Git 如何将一个二进制文件转换为文本格式，或者如何从二进制数据提取文本信息（例如元数据），然后使用普通的文本比较命令对上述文本进行差异比较。转换成文本之后会丢失不少信息，但是差异比较的结果也是非常有用的（即使并不能显示所有的差异变更）。

用户也可以通过 `textconv` 配置项作为 `diff` 命令的驱动达到上述目的。通过它用户可以为某个程序指定文件名作为执行参数，并在输出结果中显示对应的文本信息。

例如用户希望查看微软的 Word 文档文件和 JPEG 图片文件元数据版本历史的差异。首先用户需要在 `.gitattributes` 文件中添加以下配置信息：

```
*.doc    diff=word
*.jpg    diff=exif
```

用户可以使用诸如 `catdoc` 这类程序，将 Word 文档中的二进制数据转换成文本，使用 `exiftool` 从 JPEG 图片中提取 EXIF 元数据。因为转换过程可能会非常慢，Git 系统提供了一种机制，通过布尔属性 `cachetextconv` 来让用户决定是否缓存输出结果。缓存的数据是以便签形式存储的（该机制会在第 8 章详细介绍）。该配置在配置文件中对应的内容和下文类似：

```
[diff "word"]
    textconv = catdoc

# cached data stored in refs/notes/textconv/exif
[diff "exif"]
    textconv = exiftool
    cachetextconv = true
```

用户可以使用 `git show` 或者 `git cat-file -p` 加 `--textconv` 选项查看经过 `textconv` 属性过滤的输出结果。

更复杂也更强大的方案是使用 `diff` 命令的 `command` 选项指定一个外部 `diff` 驱动（属性版本的全局驱动可以通过环境变量 `GIT_EXTERNAL_DIFF` 或者 `diff.external` 配置变量指定）。换句话说，用户需要舍弃 Git 的 `diff` 命令支持的某些选项特性：着色，Word 差异比较，合并差异表比较等。

这类程序一般会调用七个参数：路径(`path`)，源文件(`old-file`)，旧进制(`old-hex`)，原模式(`old-mode`)，目标文件(`new-file`)，新进制(`new-hex`)和新模式(`new-mode`)。



这里的源文件和目标文件是指 diff 驱动可识别文件的两个不同版本，旧进制和新进制是指与文件内容相关的 SHA-1 识别码，旧模式和新模式是指文件内容的八进制表示。该命令预期的输出结果和 diff 命令类似，例如，用户希望使用兼容 XML 文件的 diff 比较工具来比较 XML 文件：

```
$ echo "*.xml diff=xmldiff" >>.gitattributes
$ git config diff.xmldiff.command xmldiff-wrapper.sh
```

本示例假定用户已经写好了 xmldiff-wrapper.sh shell 脚本，以便重新排列选项顺序兼容 XML 差异比较工具。

### 配置 diff 输出结果

Git 采用的 diff 格式显示变更记录格式描述已经在第 3 章详细介绍过。在差异比较输出结果中，每一组变更（又称变更块）都用块首部分割线区隔开来。例如：

```
@@ -18,6 +19,8 @@ int cmd_http_fetch(int argc, const char **argv,
```

第二个 @@ 之后的内容描述了变更块所处的具体位置，对于 C 源码文件来说，是它在函数的起始位置。决定如何监测变更所在位置的描述信息，可以根据文件类型进行处理。Git 允许用户通过 diff 驱动的 xfuncname 配置项进行设置，而且它还支持正则表达式匹配文件内部对应的变更部分。例如对于 LaTeX 文档，用户也许希望使用如下 tex 的 diff 驱动的配置（不过用户不一定非要这么做，因为 tex 只是 diff 驱动内置的一种预定义选项之一）：

```
[diff "tex"]
  xfuncname = "^((\\(\\(sub)*section\\{.*\\})$")
  wordRegex = "\\([a-zA-Z]+\\{\\}\\|\\(\\(\\{\\}\\{[:space:]]+\\)"
```

wordRegex 选项定义了 LaTeX 文档中执行 git diff --word-diff 命令后需要比较差异的内容。

### 三路合并

用户还可以使用 merge 属性告知 Git 系统在用户的项目中对于特定的文件或者某一类文件使用特定的合并策略。在处理文本文件时，Git 会默认采用三路合并驱动程序（类似 rcsmerge）处理它，同时处理二进制文件时会把它当作一个有冲突的合并提交结果。用户可以通过设置 merge 属性（或者使用 merge=text）强制要求 Git 以三路合并方式处理文件；也可以不设置这个属性，系统会强制使用处理二进制文件合并的方式处理文件（也可以使用 -merge 选项，而且它和 merge=binary 等效）。

用户还可以自己编写合并驱动程序，或者配置 Git 使用第三方的外部合并驱动程序。例如用户希望在自己的版本库中使用 GNU 风格的变更日志（附带描述信息的变更记录表），就可以使用 `git-merge-changelog` 命令调用 GNU 可移植库（GNULib），同时用户需要在 Git 配置文件中做如下配置：

```
[merge "merge-changelog"]
  name = GNU-style ChangeLog merge driver
  driver = git-merge-changelog %O %A %B
```

`merge.merge-changelog.driver` 中的符号 `%O` 代表包含源文件祖先修订的临时文件名，符号 `%A` 和符号 `%B` 分别代表将要合并的当前修订版本（合并所有分支后的版本）和其他分支的修订版本的临时文件名。合并驱动程序的预期目标是将以合并的修订版本存放到 `%A` 文件中，如果存在合并冲突，系统会返回非 0 状态码。

注意，用户可以使用别的驱动程序处理相同祖先（祖先只有一个）的内部合并。例如在处理二进制文件时，可以使用 `merge.*.recursive` 完成该功能。

当然，用户还需要告知 Git 系统在生成变更日志文件时调用此驱动程序，可以在 `.gitattributes` 文件中添加如下信息：

```
ChangeLog merge=merge-changelog
```

## 4.2.2 文件转换（内容过滤）

有时，用户也许希望将内容转换成 Git 系统、平台（操作系统）、文件系统和用户方便识别的格式。尾行转换规范可以认为是这类操作比较特殊的一种情况。

为此，用户需要设定相应路径的过滤属性，并且配置适当的过滤驱动程序（当然也可以不指定驱动程序）。如图 4-1 所示，当签出的文件符合某个给定的模式时，过滤命令会将该文件作为标准输入，然后将标准输出结果更新工作目录中对应的文件。

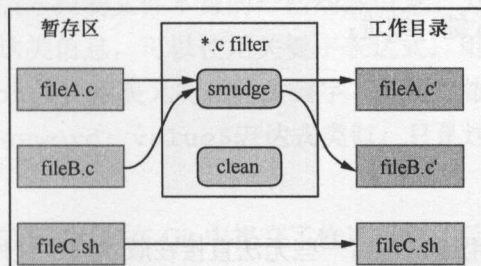


图 4-1 过滤器正在签出的文件中工作（写入文件到工作目录）

类似地，如图 4-2 所示，清洁过滤命令专门用于将工作区的文件内容转换成适合存放在版本库中的格式。

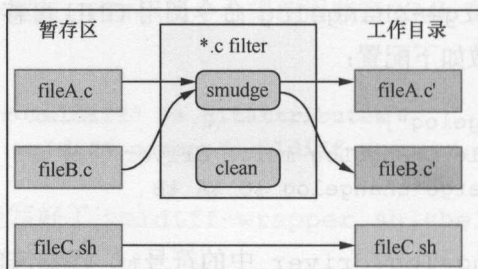


图 4-2 “清洁”过滤器在过滤暂存区的文件（添加所有到暂存区）

当声明一个命令时，用户可以使用%f符号，它可以用来替换成过滤器正在处理的文件名。

一个简单的示例是使用对 ODF（OpenDocument Format，ODF）文件进行重新压缩的脚本。ODF 文档是大部分 XML 文件的 ZIP 格式文件。Git 本身支持文件压缩，并且还支持增量化压缩（但是对已压缩的文件不起作用）；其主要思路是在版本库中存放未压缩的文件，但是签出状态的文件是经过压缩的：

```
[filter "opendocument"]
  clean = "rezip -p ODF_UNCOMPRESS"
  smudge = "rezip -p ODF_COMPRESS"
```

当然，用户还需要告知 Git 希望对所有 ODF 文件都执行内容过滤：

```
*.odt filter=opendocument
*.ods filter=opendocument
*.odp filter=opendocument
```

另外一个比较常用的过滤器是对代码格式进行强制重排版。常见的示例是迁入代码时使用空格替换制表符（Tab 键）缩进：

```
[filter "indent"]
  clean = indent
```

## 强制文件转换

内容过滤的另外一个用途是存储一些无法直接在版本库中使用的内容，然后在将它们签出时转换为可用格式。



这样的示例就有诸如使用 `gitattributes` 存储大型的二进制文件。在 Git 版本库外部，开发人员只需要用到该文件中的一小部分；而在版本库内部，只需要提供一个标识符，方便用户获取外部存储的内容即可。这就是 `git-media` 的工作原理：

```
$ git config filter.media.clean "git-media filter-clean"
$ git config filter.media.smudge "git-media filter-smudge"
$ echo "*.mov filter=media -crlf" >> .gitattributes
```



用户可以在 <https://github.com/schacon/git-media> 找到 `git-media` 多媒体工具。其他类似的工具在第 9 章会详细介绍。

另外一个示例是加密敏感信息，或者使用占位符替换某个应用程序必需的本地配置信息（例如数据库密码）。因为运行这样一个过滤器需要获取一些有用的信息，用户可以做如下配置：

```
[filter "clean-password"]
  clean = sed -e 's/^pass = .*$/pass = @PASSWORD@/'
  smudge = sed -e 's/^pass = @PASSWORD@/pass = passw0rd/'
  required
```

注意，这只是一个非常简单的示例，在实际生活中，用户还需要考虑配置文件本身的安全性，或者将实际的密码存放在一个外部的混淆脚本中。这种情况下，用户最好也建立一个预提交、预拉取和更新的钩子，以便确保密码不会提交到公共版本库中（详情可以参考第 10 章）。

### 4.2.3 关键字替换表达式

用户有时需要获取版本控制文件本身的一些动态信息。为了确保用户在访问版本控制系统时能够及时获取这类信息，可以使用关键字表达式：用文本替换关键字锚点，其形式类似这样：`$Keyword$`，用美元符号将关键字（关键字锚点）包裹起来。它的原理和版本控制系统中的 `$Keyword: value$` 表达式类似，只是这种形式是关键字紧跟在表达式后面。

这样做的主要原因在于，用户在 Git 中提交了修订对象之后，因为 Git 的原因，无法再对它进行修改（详情可以参考第 8 章）。这意味着关键字锚点在版本库中的状态是只读的，只有在工作目录中处于签出状态时才能进行扩展。不过这也是一个优点，用户在查看历史



提交记录时，不会看到因为关键字表达式而产生的虚假的变更。

Git 系统内部唯一支持的关键字表达式是`$Id$`，它的值是文件内容的 SHA-1 码标识符（表示文件内容的 blob 对象 SHA-1 校验码和文件的 SHA-1 码是不同的，详情可以参考第 8 章）。用户可以设置文件的 `ident` 属性访问这个关键字表达式。

用户还可以自己编写支持相应过滤器的关键字表达式，定义清理工作目录命令的表达式扩展，并通过关键字锚点替换关键字表达式的内容，对项目进行清理。

利用这种机制，用户可以方便地完成某些任务，例如有一个关键字表达式`$Date$`，它可以在文件最近一次修改日期的基础上显示签出状态下的日期：

```
[filter "dater"]
  clean = sed -e 's/\\\$Date[^\$]*\\\$/\\\$Date\\$/'
  smudge = expand_date %f
```

扩展日期脚本可以通过将文件名作为执行参数，执行 `git log --pretty=format:"%ad" "$1"` 命令后，获得替换后的值。

用户还需要注意另外一个问题。理论上来说，Git 为了更好地执行性能，不会去访问那些没有发生变更的、将要提交的、与分支切换（签出状态）和分支回退（重置）相关的文件。这意味着关键字替换技巧不适用于项目最新修订记录的日期（和已变更文件的最新修订不同）。

如果用户在发布源代码时需要用到这类信息（例如当时提交的描述、预览版标签的发布时间），既可以将它们作为编译系统的一部分，也可以在执行 `git archive` 命令时进行关键字替换。后者是一个很常用的功能：如果文件的 `export-subst` 属性做了相应设置，Git 系统将会在添加文件到档案中时，将`$Format:<PLACEHOLDERS>$`当作通用的关键字表达式。



元关键字表达式`$Format$`的意义取决于修订标识符的有效性，用户无法强行指定。例如，在执行 `git archive` 命令时指定一个树对象的 SHA-1 标识符。

占位符和`--pretty=format`的作用一样：用户可以为 `git log` 命令指定自己喜欢的格式，详情可以参考第 2 章。例如字符串`$Format:%H$`将会被提交记录的哈希码替换（未展开）。这是一种不可逆的关键字替换过程，在归档（导出）操作过程中并没有对关键字进

行跟踪记录。

## 4.2.4 其他内置属性

用户还可以告知 Git 在文件归档时忽略特定文件或者目录。例如用户也许希望在发布测试时，将包含用户文件信息的文件目录排除在外，这一点对于开发人员和最终用户（他们也许还需要额外的工具，或者优先考虑软件的质量和特性，而非纠正软件的错误）都是非常有用的。用户可以设置 `export-ignore` 属性达到上述目的，例如在 `.gitattributes` 文件中添加如下内容：

```
xt/ export-ignore
```

此外，还可以通过文件属性配置定义内容差异以及特定文件如何处理空白符错误。上述功能是配置变量 `core.whitespace` 细粒度版本控制功能的具体应用。注意，在处理常见的空白符问题时，要特别留意使用逗号作为元素分隔符的情况，在 `.gitattributes` 文件中使用该分隔符时，它的两边没有空格。具体示例如下（摘自 Git 项目）：

```
*      whitespace=!indent, trail, space
*.[ch] whitespace=indent, trail, space
*.sh   whitespace=indent, trail, space
```

用户还可以通过文件属性声明给定文件的字符编码格式，即提供一个编码属性值。Git 可以用它来选择文件在 GUI 工具中的显示形式（例如 `gitk` 和 `git gui`）。这一功能可以通过配置变量 `gui.encoding` 进行细粒度的控制，并且可以基于性能因素进行显式声明。例如 GNU 的 `gettext` 可移植性对象文件希望使用 UTF-8 格式进行编码，那么可以做如下配置：

```
/po/*.po encoding=UTF-8
```

## 4.2.5 属性宏定义

在识别二进制文件和换行符约定章节中，我们已经学习了如何使用二进制属性标记二进制文件。这实际上是 `-diff -merge -text`（未设置的 3 个文件属性）3 个选项的属性宏表达式。有更优雅的方式定义这类宏，并且可以避免不必要的重复。还可以使用多个模式匹配给定的文件类型，不过 `gitattribute` 属性文件中一行只能包含一种文件模式。Git 允许用户定义这类宏，不过仅限于顶层的 `gitattributes` 文件 `core.attributesFile`、`.git/info/attributes` 或者项目主目录下的 `.gitattributes`

文件。系统内置的二进制宏的定义如下：

```
[attr]binary -diff -merge -text
```

用户还可以定义自己的属性，然后编写程序，检查给定文件都设置了哪些属性；或者使用 `git check-attr` 命令查看某类文件的属性值是什么。

## 4.3 使用 reset 命令修复错误

在开发过程的任意阶段，用户也许希望撤销一些东西以便达到修复错误的目的，或者希望将当前的工作成果丢弃。在 Git 核心中不存在 `git undo` 这样的命令，也没有任何命令存在 `--undo` 选项可供使用。与此同时，倒是很多命令中都有 `--abort` 选项，可以让用户丢弃当前的工作成果。没有这类命令或者选项的原因之一就是为了避免在被撤销内容上产生歧义（这对于多步操作尤其重要）。

很多问题都可以通过 `git reset` 命令进行修复。它的用途非常广泛，理解它的工作原理会让读者在实际应用中如虎添翼。

注意，本章节的内容只介绍了在完整树结构下使用 `git reset` 命令的情况。重置文件状态使用的是 `git reset -- <file>` 命令，它的具体使用将会在本章后续的章节详细介绍。

### 4.3.1 回退分支 head

`git reset` 命令在完整的树模式下不仅会影响当前分支首部，而且还会影响索引（暂存区）和工作目录。注意，重置不会影响当前的分支，只会如第 3 章所述，对签出状态下的内容产生影响。为了只重置当前分支首部，不影响所有工作目录，可以使用 `git reset --soft [<revision>]` 命令。

从效率上来看，我们只修改了当前分支的指针（如图 4-3 所示的 master 分支）指向了一个给定的修订节点（HEAD<sup>^</sup>代表本示例中的上一提交记录），暂存区和工作区都没有受到影响。这一操作使得用户丢弃了将要提交状态中所有已变更的文件（分支上与上一版本存在差异文件），使用 `git status` 命令后会显示这一变化。

#### 移除和修改提交

上述命令的工作方式意味着软性重置可以用来撤销创建提交记录的行为。这也适用于提交的修改，这种方式要比 `git commit` 命令和 `--amend` 选项一起使用更简单一些。事



实上, 执行下列命令:

```
$ git commit --amend [<options>]
```

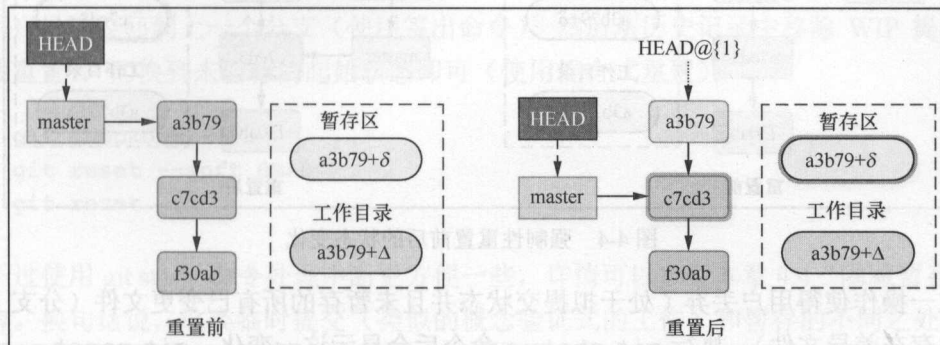


图 4-3 强行重置前后的区别

和执行下列命令是等效的:

```
$ git reset --soft HEAD^
$ git commit --reedit-message=ORIG_HEAD [<options>]
```

和使用软性重置不同之处在于, `git commit --amend` 命令也适用于合并提交。在修改提交时, 如果用户只想修改提交的注释信息, 那么不需要使用任何命令选项辅助。如果用户想修复一个工作目录中的问题, 但是不想修改注释信息, 那么可以使用 `-a --no-edit` 选项。如果用户希望修复用于纠正 Git 配置项目中的作者信息, 那么可以使用 `--reset-author --no-edit` 选项。

## 使用 reset 压缩提交记录

用户并不仅限于将分支的首部移动到上一个提交节点。通过软性重置, 用户还可以插入几个更早的提交记录 (例如提交和 bug 修复, 或者引入新的功能函数), 将一个提交一分为二 (或者更多), 又或者使用 `squash` 指令进行交互式变基操作, 详情可以参考第 8 章。对于后者, 用户实际上可以插入任意一系列的提交记录, 而且不局限于最近的几个提交记录。

## 4.3.2 重置分支 head 和索引

`reset` 命令的默认模式也被称为混合式重置 (因为它是介于软性重置和强制重置之间的), 修改当前分支的首部指向给定的修订节点, 同时重置索引, 将相关修订的内容写入暂存区 (见图 4-4)。



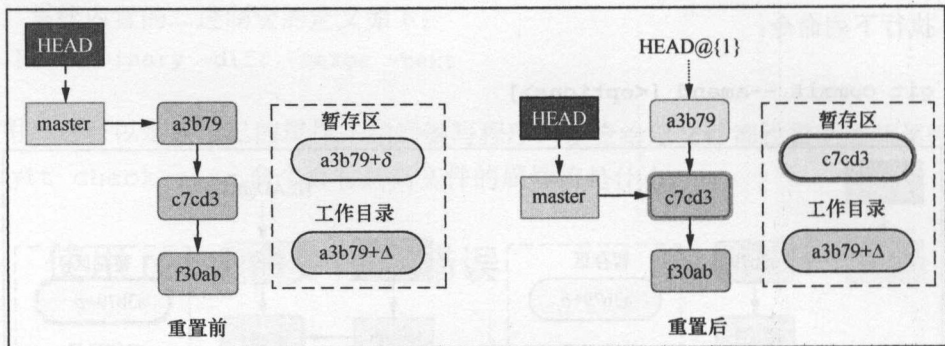


图 4-4 强制性重置前后的状态变化

这一操作使得用户丢弃了处于拟提交状态并且未暂存的所有已变更文件（分支上与上一版本存在差异文件），执行 `git status` 命令后会显示这一变化。`git reset --mixed` 命令还会使用简要状态格式报告那些未更新的内容。用户还可以访问 `reset` 命令的历史版本，例如撤销新增的文件。如果用户没有暂存任何变更的话（或者用户可以直接丢弃这些变更），那么上述操作可以使用 `git reset` 命令实现。如果用户希望撤销添加某个特定文件的操作，那么可以使用 `git rm --cached <file>` 命令。

### 使用 reset 分割提交

用户还可以使用混合式的 `reset` 命令将一个提交一分为二。首先，运行 `git reset HEAD^` 命令，将分支首部和索引指向上一个修订节点。然后向第一个提交以交互式添加的方式添加用户希望添加的变更，继而根据索引创建第一个提交（`git add -i` 和 `git commit`）。第二个提交可以根据工作目录状态创建（`git commit -a`）。

如果交互式移除变更时更简便一些，那么用户也可以这么做。使用 `git reset --soft HEAD^` 命令后，可以对每个文件执行 `reset` 命令，实现交互式地撤销暂存变更，根据索引的构造状态创建第一个提交记录，然后根据工作目录创建第二个提交记录。

依次循环往复操作，用户就可以替代交互式变基，实现分割历史提交记录的目的。变基操作会切换到适当的提交记录上，实际的分割操作大致和上述操作是一样的。

### 使用 WIP 提交保存和重排变更状态

假如你正在某个分支上进行功能开发工作，这时有一个紧急的 bug 修复请求使得你不得不中断目前的工作。你又不想舍弃目前分支上产生的变更记录，但是现在的工作目录又有些凌乱。一个比较可行的解决方案是通过创建一个临时提交（工作进行中的快照，WIP），以便保存当前工作区的状态：

```
$ git commit -a -m 'snapshot WIP (Work In Progress)'
```

然后用户就可以停下目前的工作，切换到维护分支，创建一个提交专门修复相关问题。之后用户只需要回到上一个分支（使用签出命令），然后从历史记录中移除 WIP 提交（使用软性重置），切换到未暂存的起始状态即可（使用混合式重置）：

```
$ git checkout -
$ git reset --soft HEAD^
$ git reset
```

不过使用 `git stash` 命令处理中断更方便一些，详情可以参考本章 4.4 “隐藏暂存变更”的内容。换句话说，这类临时提交（类似的概念验证式的工作）和暂存的不同之处在于，它可以和团队其他成员共享。

### 4.3.3 丢弃变更和回退分支

有时看着一团乱麻的工作目录，用户也许非常希望丢弃所有的变更，将工作目录和暂存区（索引）的状态恢复到最近一次提交的状态（最新的稳定版本程序）。又或者用户希望将版本库的状态回退到以往的某个修订版本。如图 4-5 所示，强制性重置（reset）操作将会修改当前分支首部的指向，并且重置索引和工作目录树。被跟踪文件产生的所有变更都会被丢弃。

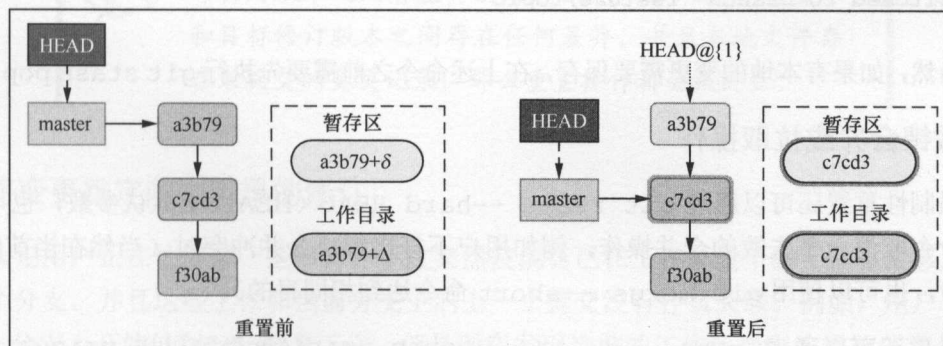


图 4-5 强制性重置前后的状态变化

该命令还可以用于撤销（移除）一个提交，就好像该提交从未存在过一样。执行 `git reset --hard HEAD^` 命令后，将会高效地将上一个提交节点移除（不过短时间内还可以通过 `reflog` 恢复），除非该节点还可以通过其他分支进行访问。

另外一个常见的用法是通过 `git reset --hard` 命令丢弃工作目录下的变更。



特别重要的一点需要谨记：强制性重置操作是不可恢复地将暂存区和工作目录中的所有变更移除。用户无法撤销这部分操作！变更记录将会永远消失！

## 将提交指向特性分支

假如用户正在 `master` 分支上工作，而且已经创建了一系列的提交，同时发现正在研发的功能特性之间联系非常紧密，继而希望将它们整合到一个主题分支上。主题分支的详情可以参考第6章。用户希望将在 `master` 分支上的一系列提交（例如最近的3个修订版本）迁移到上述特性分支上。

用户需要创建一个 `feature` 分支，保存为提交的变更（如果有的话），将 `master` 分支上与主题特性相关的提交记录移除，最后切换到 `feature` 分支继续工作（或者可以使用变基操作实现）：

```
$ git branch feature/topic
$ git stash
No local changes to save
$ git reset --hard HEAD~3
HEAD is now at f82887f before
$ git checkout feature/topic
Switched to branch 'feature/topic'
```

当然，如果有本地的变更需要保存，在上述命令之前需要先执行 `git stash pop` 命令。

## 撤销合并或拉取操作

强制性重置还可以通过 `git reset --hard HEAD`（`HEAD` 是默认参数，也可以省略）命令取消一个失败的合并操作，例如用户不打算解决合并冲突时（当然在当前的 Git 系统中，也可以使用 `git merge --abort` 命令达到相同目的）。

用户还可以通过 `git reset --hard ORIG_HEAD`（这里可以用 `HEAD@{1}` 替代 `ORIG_HEAD`）命令移除一个成功执行的快进式拉取操作或者变基操作（以及其他移动分支首部的操作）。

## 4.3.4 安全模式重置——保留用户变更

强制性重置操作会丢弃用户本地的所有变更，它的效果和 `git checkout -f` 命令类



似。有时用户也许希望在回退当前分支的同时保留本地的变更,这就是 `git reset --keep` 命令能够实现的功能 (见图 4-6)。

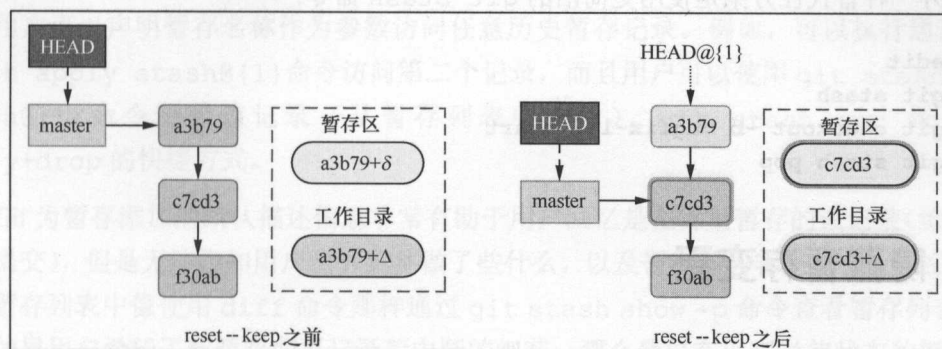


图 4-6 执行 `git reset --keep HEAD^` 命令前后的状态变化

这种模式会重置暂存区(索引实体),但是仍然保留当前本地工作目录下未暂存的变更。如果遇到异常,该重置操作将会被终止。这意味着工作区中的变更被保存了,并且被移动到了新的提交记录中,其工作原理和 `git checkout <branch>` 命令处理未提交的变更记录类似。执行成功的情况和隐藏变更类似,强制性重置,然后取消隐藏变更。



`git reset --keep <revision>` 命令的工作原理是更新(工作目录下)我们回退的目标版本和分支首部引用版本之间有差异的文件内容。如果在分支首部和目标修订版本之间存在任何差异,并且本地文件存在未提交的变更记录,那么重置操作都会被终止。

## 将变更重定向到较早的修订

假定用户正在忙着做一些事情,不过突然发现自己在工作目录下的工作本应该属于另外一个分支,并且这些工作和当前分支上的上一个提交没有什么关联。例如,用户也许在 `master` 分支上开始处理 bug 修复工作,但是现在发现当前的工作还会影响维护分支 `maint`。

这意味着 bug 修复的提交记录应该被加入更早的分支,起点可能是上述分支的共同祖先节点(或者是引入 bug 的某个位置)。这可能会导致 `master` 和 `maint` 分支合并相同的 bug 修复提交节点,详情可以参考第 12 章:

```
$ edit
$ git checkout -b bugfix-127
```



```
$ git reset --keep start
```

另外一种替代性方案是使用更简洁的 `git stash` 命令：

```
$ edit
$ git stash
$ git checkout -b bugfix-127 start
$ git stash pop
```

## 4.4 隐藏暂存变更

一般来说，计划赶不上变化，当用户参与到一个项目中来时，经常需要临时保存一下当前的工作状态，然后处理其他的工作。`git stash` 命令是处理这类问题的好帮手。

暂存操作会保存用户凌乱无章的工作目录状态，该状态是指用户工作目录下已经发生变更的被跟踪文件（当然，用户还可以使用 `--include-untracked` 选项暂存未跟踪的文件）以及暂存区的状态，系统保存该状态之后，通过运行 `git reset --hard HEAD` 命令，将会重置工作目录和索引，回退到最近一次提交的修订版本（为了匹配首部提交）。用户之后还可以随意地访问已经暂存的变更记录。

暂存记录是保存在堆栈上的：默认情况下，用户读取的是最后一次入栈的暂存变更（`stash@{0}`）。当然，用户还可以查看暂存变更列表（使用 `git stash list` 命令），并且可以显式访问某个特定的暂存变更记录。

### 4.4.1 使用 `git stash`

如果用户不希望被其他工作打断太久，那么可以简单地将目前的工作暂存起来，处理完别的事务之后，再将暂存的记录恢复即可：

```
$ git stash
$ ... handle interruption ...
$ git stash pop
```

默认情况下，`git stash pop` 命令会恢复最后一次暂存的变更记录，如果恢复成功，那么该记录将会从堆栈中删除。若希望查看用户的暂存记录列表，可以使用 `git stash list` 命令：

```
$ git stash list
```

```
stash@{0}: WIP on master: 049d078 Use strtol(), atoi() is deprecated
stash@{1}: WIP on master: c264051 Error checking for <number>
```

用户可以声明暂存名称作为参数访问任意历史暂存记录。例如，可以执行通过 `git stash apply stash@{1}` 命令访问第二个记录，而且用户可以使用 `git stash drop stash@{1}` 命令删除该记录（从暂存列表中删除）。`git stash pop` 命令只是 `apply+drop` 的快捷方式。

Git 为暂存添加的默认描述信息非常有助于用户回忆是在哪里暂存的该记录（给定分支或者提交），但是无法告知用户当时具体做了些什么，以及暂存的内容是什么。不过用户可以在暂存列表中像使用 `diff` 命令那样通过 `git stash show -p` 命令查看暂存列表细节。不过如果用户希望了解暂存变更记录后中断的细节，那么最好在保存当前状态的暂存记录中附加更详细的描述信息：

```
$ git stash save 'Add <count>'
Saved working directory and index state On master: Add <count>
HEAD is now at 049d078 Use strtol(), atoi() is deprecated
```

Git 将会使用用户提供的信息描述已暂存的变更：

```
$ git stash list
stash@{0}: On master: Add <count>
stash@{1}: WIP on master: c264051 Error checking for <number>
```

有时候，当用户在目前工作的分支上执行 `git stash save` 后，因为该分支上新增了太多变更，以致于出现无法顺利执行 `git stash pop` 命令的现象，这主要是因为用户暂存变更后，新增了不少基于该修订的修订版本。如果用户希望在暂存变更之外再新建一个常规的提交，或者只是希望测试一下暂存的变更，那么可以使用 `git stash branch <分支名>` 命令。这会在用户保存变更的那个修订版本的基础上新建一个分支并切换到该分支，恢复用户之前保存的变更，然后将上述暂存的变更在暂存列表中删除。

## 4.4.2 隐藏和暂存区

默认情况下，暂存操作会重置工作目录和暂存区状态到 HEAD 引用的版本。用户可以使用 `git stash` 命令保留索引的状态，然后通过使用 `--keep-index` 选项将工作区重置到暂存状态。

这一点非常有用，特别是当用户遇到第 3 章 3.1 “新建提交”中的情况，使用暂存区

处理工作区不规则的变更记录时；又或者用户希望如本章前文所述，希望将提交一分为二时。在上述两种情况下，用户都应该在提交变更之前对每个变更进行测试。这种工作流程如下：

```
$ git add --interactive
$ git stash --keep-index
$ make test
$ git commit -m 'First part'
$ git stash pop
```

用户还可以使用 `git stash --patch` 命令在将变更暂存之后，指定工作区的表现形式。

在恢复暂存变更时，Git 系统一般会尝试只恢复工作区的暂存变更，然后将它们和当前工作目录状态整合（例如和暂存区的内容对应）。如果在整合过程中出现冲突，这些记录会被当作普通的索引存储到暂存区中，即使存在冲突，Git 系统也不会丢弃暂存记录。

用户还可以使用 `--index` 选项恢复暂存区被隐藏的暂存记录，如果记录之间存在冲突，整合操作将不会成功执行（因为暂存区没有地方为这种冲突记录提供存储空间）。

### 4.4.3 暂存探幽

也许用户恢复了某些暂存变更之后，完成了某些工作，由于某些原因又希望撤销恢复的暂存变更。或者用户因为失误将部分暂存内容删除了，又或者清空了所有暂存记录（可以使用 `git stash clear` 命令），现在希望恢复这些记录。又或者用户想看看暂存变更之后工作目录中的文件组织结构。例如，用户需要知道当创建一个暂存记录后，Git 系统内部到底执行了哪些操作。

为了暂存用户的变更记录，Git 系统会自动创建两个提交对象：一个是和索引有关的（暂存区），另一是和工作目录有关的。通过 `git stash --include-untracked` 命令，Git 系统会另外为未跟踪文件自动创建提交对象。

提交对象中包含工作目录下的工作进度，即暂存对象，并且将包含暂存区内容的提交对象作为其第二个父对象。提交对象中存放在了一个特别的引用中：`refs/stash`。工作中（WIP）和索引的提交对象中都包含用户保存变更时的修订版本，并且将它当作第一个（仅对于索引提交对象来说）父对象。

我们可以使用 `git log --graph` 命令或者 `gitk` 图形化工具查看它们：



```
$ git stash save --quiet 'Add <count>'
$ git log --oneline --graph --decorate --boundary stash ^HEAD
* 81ef667 (refs/stash) On master: Add <count>
|\
| * ed95050 index on master: 765b095 Added .gitignore
|/
o 765b095 (HEAD, master) Added .gitignore
$ git show-ref --abbrev
765b095 refs/heads/master
81ef667 refs/stash
```

这里我们不得不使用 `git show-ref` 命令（我们还可以使用 `git for-each-ref` 替代），这是因为 `git branch -a` 命令只显示分支信息，但是不显示相关的引用信息。

当保存未跟踪变更记录时，情况和下列步骤类似：

```
$ git stash --include-untracked
Saved working directory and index state WIP on master: 765b095 Added\
.gitignore
HEAD is now at 765b095 Added .gitignore
$ git log --oneline --graph --decorate --boundary stash ^HEAD
*-. bb76632 (refs/stash) WIP on master: 765b095 Added .gitignore
|\ \
| | * 1ae1716 untracked files on master: 765b095 Added .gitignore
| * d093b52 index on master: 765b095 Added .gitignore
|/
o 765b095 (HEAD, B) Added .gitignore
```

我们可以看到，未跟踪文件提交是 WIP 提交对象的第三个父提交，而且它没有父提交。这就是暂存的工作原理，但是 Git 系统如何维护暂存栈呢？

如果你之前注意过 `git stash` 命令的输出结果，其中的 `stash@{<n>}` 表达式和 `reflog` 的类似，那么你应该已经猜到了，Git 在引用日志中查找旧的暂存记录的方式是通过 `refs/stash` 引用实现的：

```
$ git reflog stash
81ef667 stash@{0}: On master: Add <count>
bb76632 stash@{1}: WIP on master: Added .gitignore
```

## 暂存记录撤销

接下来将会演示本小节的第一个示例：撤销以前执行 `git stash apply` 命令的操作



结果。一个可以满足上述需求的备选解决方案是修改暂存中和工作目录变更有关的补丁，然后反向应用它：

```
$ git stash show -p stash@{0} | git apply -R -
```

注意，`git stash` 命令的 `-p` 选项会显示强制补丁而非变更摘要。我们可以使用 `git show -m stash@{0}` 命令（`-m` 选项是必需的，因为 WIP 提交在暂存中是以合并提交的形式存在的），或者也可以简单地使用 `git diff stash@{0}^1 stash@{0}` 命令替代 `git stash show -p`。

### 恢复误删除的暂存

接下来将演示第二个示例：恢复误删除的暂存记录。如果它们仍然在版本库中，用户可以通过引用搜索所有不可达的提交对象和类似的暂存记录（它们使用严格模式并且附带了一个注释信息的合并提交对象）来恢复。

一个简化版的解决方案可能是这样的：

```
$ git fsck --unreachable |  
grep "unreachable commit " | cut -d" " -f3 |  
git log --stdin --merges --no-walk --grep="WIP on "
```

第一行是找到所有不可达的对象，第二行是过滤除了提交和与之对应的 SHA-1 码标识符之外的所有信息，第三行的意思是进一步过滤，只显示注释中包含 "WIP on " 的合并提交记录。

不过这个方案并不是完美无缺的，例如查找一个自定义注释信息的暂存记录（该记录是通过 `git stash save "信息"` 命令创建的）。

## 4.5 管理工作区和暂存区

第3章已经介绍过，除了用户的工作区和存放提交记录的本地版本库之外，它们之间还有一个部分，即暂存区，有时也称索引。

在上述章节中，我们还学习了如何查看工作目录的状态，比较版本之间的差异等；同时还学习了如何在工作区或者暂存区之外创建一个新的提交对象。

现在我们将要学习如何查看和修改单个文件的状态。

## 4.5.1 查看文件和目录

查看工作目录中的内容很简单：只需要使用标准的文件浏览工具即可（例如一个编辑器或者文件浏览器，也可以使用 `dir` 命令）。不过如何查看某个文件在暂存区的内容或者最近的修订版本？

一个可行的方案是使用 `git show` 命令搭配适当的过滤条件。在第 2 章中，我们已经学过通过 `<修订>:<路径名>` 语法来查看给定修订的文件内容。类似的语法也可以用于查看暂存内容，即 `:<路径名>`（如果文件中存在合并冲突，那么也可以使用 `:<暂存>:<路径名>`，`:<pathname>` 本身是和 `:0:<pathname>` 等价的）。

现在假定用户当前处于 `src/` 子目录下，希望查看工作目录中的文件 `rand.c` 在暂存区中（使用绝对和相对路径）的内容，以及它最近一次提交的记录：

```
src $ less -FRX rand.c
src $ git show :src/rand.c
src $ git show ../rand.c
src $ git show HEAD:src/rand.c
src $ git show HEAD:../rand.c
```

为了获知哪些文件被暂存到索引中，可以使用 `git ls-files` 命令。默认情况下，该操作的目标是暂存区中的内容，不过它也可以用来查看工作区的情况（本章前面的内容中已经使用该命令查看被忽略的文件）。这个命令会列出特定目录或者当前目录下的所有文件（因为索引是文件的线性列表，类似于 `MANIFEST` 文件）。用户还可以使用 `/` 表示项目的顶层目录。如果没有指定 `--full-name` 选项，查询结果将只显示相对当前目录（或者用参数指定的文件目录）的文件名。在上述两个示例中，我们都假定处于 `src/` 目录下，相应的命令提示符中的结果如下：

```
src $ git ls-files
rand.c
src $ git ls-files --full-name :/
COPYRIGHT
Makefile
README
src/rand.c
```

那么如何查看已提交的变更呢？如何查看给定修订对应的文件？用户可以使用 `git ls-tree` 命令达到上述目的（注意这是一个底层命令，默认情况下不会直接访问 `HEAD`

修订):

```
src $ git ls-tree --name-only HEAD
rand.c
src $ git ls-tree --abbrev --full-tree -r -t HEAD
100644 blob 862aafd    COPYRIGHT
100644 blob 25c3d1b    Makefile
100644 blob bdf2c76    README
040000 tree 7e44d2e    src
100644 blob b2c087f    src/rand.c
```

## 4.5.2 搜索文件内容

现在假定用户正在审核项目代码，并且发现 C 源代码中出现了包含 ';;' 符号的错误。或者用户在编辑代码文件时发现了一个错误。当你修复这些错误之后，心里一定会想：“这类错误还有多少没被发现？”接下来用户很可能需要为修复这类错误创建一个错误修复提交。又或者用户希望查找一些和下次将要提交的修订相关的信息？又例如用户希望查看上述提交在 next 分支上的分布情况？

在 Git 系统中，用户可以使用 `git grep` 命令达到上述目的：

```
$ git grep -e ';;'
```

上述命令会在工作目录的当前目录向下搜索所有被跟踪文件。我们将会得到很多误报结果，例如在 shell 脚本中，因此可以将搜索范围限制在 C 源代码文件中：

```
$ git grep -e ';;' -- '*.c'
```

为了替代 `git grep` 命令通过 shell 获取文件扩展列表，上述单引号对于 Git 功能扩展来说是必需的（路径约束）。当然我们还会从 C 语言的死循环代码块中获得一些错误的匹配结果：

```
for (;;) {
```

使用 `git grep` 命令，用户可以构造复杂的条件表达式，将误报结果排除。如前所述，当我们希望搜索整个项目而不是当前目录时：

```
$ git grep -e ';;' --and --not 'for *(*;;' -- '**/*.c'
```

为了搜索暂存区，用户可以使用 `git grep --cached` 命令（也可以使用 `git grep`

--staged 命令达到同样的目的)。为了搜索下一个分支，可以使用 `git grep next --` 命令。实际上，类似的命令还可以用于搜索任意修订版本。

## 4.5.3 撤销对文件的跟踪、暂存和修改

如果用户希望撤销某些文件级的操作（例如用户希望改变被跟踪文件的内容或者某些暂存区的变更），那么最好先使用 `git status` 命令看看项目中发生了哪些变更：

```
$ git status --ignored
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    changes to ...

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working\
directory)

    changes to ...

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    ...

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
```

用户需要谨记的是，只有工作目录和暂存区的内容是可以编辑的，已提交的变更是无法更改的。如果用户希望撤销添加未跟踪文件到索引的操作，或者从暂存区移除一个被跟踪文件，以便在下一个提交中移除该文件，那么可以在保持工作目录状态的同时，使用 `git rm--cached <file>` 命令达到此目的。



### --cached (--staged)和--index 的区别

大部分 Git 命令，例如 `git diff`、`git grep` 和 `git rm`，都支持使用 `--cached` 选项（以及它的别名 `--staged`）。此外，像 `git stash` 这样的命令，还支持 `--index` 选项（索引是暂存区的代名词）。不过它们并不是同义词（因为稍后我们将会看到，`git apply` 命令同时支持上述两个选项的使用）。

`--cached` 选项的用途是让平时处理工作目录中文件的命令处理暂存区的文件。例如 `git grep --cached`





命令将会搜索暂存区中的内容，而不是工作目录中的内容。`git rm --cached` 命令将会只把文件从暂存区中移除，而不影响工作目录。

`--index` 选项是用来要求平时处理工作目录文件的命令，在执行的同时还要额外操作与之相关的暂存区。例如 `git stash apply --index` 命令，它不仅会还原暂存的工作目录变更，还会恢复与之相关的索引。

如果你希望 Git 系统在暂存区中记录某个路径的状态变化，但是之后又改变了主意，那么可以使用 `git reset HEAD -- <file>` 命令将暂存的文件内容重置到以前的修订版本。

如果你因为操作失误编辑了某个文件，继而导致工作区的版本发生了混乱，那么可以使用 `git checkout -- <file>` 命令从索引中将工作区的状态恢复到以前的版本。如果用户暂存的某个状态非常混乱，也可以使用 `git checkout HEAD -- <file>` 命令，将该状态恢复到最近一次提交的状态。



实际上这些命令并没有真的执行撤销操作，它们恢复到上一状态是基于工作区、索引和已提交版本的备份。例如，如果用户已经暂存了一些变更，修改了一个文件，然后将这些修改添加到暂存区中，这时用户可以重置索引到的已提交的修订版本，但是不会受到状态变更和 `git add` 命令的影响。

#### 4.5.4 文件版本回退

当然，用户还可以根据任意修订版本对单个文件进行重置或者签出。例如，用户希望使用以前的某个修订版本文件替换当前工作区的 `src/rand.c` 文件，那么可以使用 `git checkout HEAD^ -- src/rand.c` 命令（或者用 `git show HEAD^:src/rand.c` 命令将输出结果重定向到某个文件）。为了将 `next` 分支上的版本添加到暂存区，用户可以执行 `git reset next -- src/rand.c` 命令。

注意：`git add <file>`、`git reset <file>` 和 `git checkout <file>` 都支持 `--patch` 选项使用交互模式输入一个给定文件名。上述命令也可以用来手动编辑某个

文件的暂存区或者工作目录的版本，以便达到指定哪些变更应该被提交的目的（或者取消应用）。



用户也许还需要在文件名前面输入--符号，例如你有一个和分支同名的文件。

### 4.5.5 清理工作区

工作目录中未跟踪文件和目录也许会不知不觉地越来越多，它们有可能是合并操作后的一流文件或者临时文件，又或者是某些验证概念产生的工作文件，甚至可能是因为失误而生成的冗余文件。总之，这些文件实际上是杂乱无章的，而且用户也不希望 Git 系统忽略它们（详情可以参考本章的 4.1 “忽略文件”）。如果用户只是希望删除它们，那么可以使用 `git clean` 命令。

未跟踪文件在版本库中并没有备份，因此用户无法撤销对它们的删除操作（除非操作系统或者文件系统支持还原），所以建议用户首先使用 `--dry-run / -n` 选项对上述文件进行检查。在实际情况下，默认删除操作还需要使用 `--force / -f` 选项。

```
$ git clean --dry-run
Would remove patch-1.diff
```

Git 系统会从当前目录开始递归清除所有未被跟踪的文件。用户还可以将指定路径作为参数，限定系统清理文件的范围，同时还可以使用 `--exclude=<pattern>` 选项排除某些类型的文件。用户甚至可以使用 `--interactive` 选项交互式地选择需要删除的未跟踪文件。

```
$ git clean --interactive
Would remove the following items:
src/rand.c~
screenlog.0
*** Commands ***
1: clean          2: filter by pattern  3: select by numbers
4: ask each       5: quit           6: help
What now>
```

`clean` 命令也支持用户只删除被忽略的文件，例如，可以使用 `-X` 选项在移除编译产品的同时，还可以手工指定被跟踪文件（一般来说，最好将会产生编译副产品的编译系统隔

离开来, 这样在清理项目工作文件时就不需要克隆整个版本库了)。

用户还可以将 `git clean -x` 命令和 `git reset --hard` 命令结合起来一起使用, 可以创建一个原始的工作目录来测试干净的构建过程, 在此过程中将会移除忽略文件以及未忽略但是也未跟踪的文件, 同时还会将跟踪文件重置到最近的修订提交版本。



#### 脏工作目录

如果工作目录与已提交和已暂存版本的状态是一致的, 那么系统会认为它是干净的, 否则就会认为它是脏的。

## 4.6 多工作目录

Git 系统很早以前就支持通过 `git --git-dir=<path> <command>` 或者 `GIT_DIR` 环境变量声明版本库 (`.git` 目录) 的管理区位置了, 这样做的好处是能够方便地在脱机工作目录下工作。

为了能够可靠地使用多个工作目录共享同一版本库, 用户必须采用 Git 2.5 及以上版本的软件。采用该版本的软件之后, 用户可以使用 `git worktree add <path> <branch>` 创建一个带链接的工作目录树, 而且允许用户签出多个分支。为了方便起见, 如果用户省略了 `<branch>` 参数, 系统会自动在已有的工作树上创建一个新的分支。



如果用户使用的 Git 版本比较老旧, 那么还可以使用 `git-new-workdir` 脚本创建新的目录, 该脚本位于 Git 项目版本库下的 `contrib/area` 文件夹下。不过它只适用于 UNIX 系统 (它依赖符号链接), 而且有些不太稳定。

如果用户希望在不同分支间切换, 上述机制可以替代 `git stash` 命令 (例如修复紧急安全漏洞), 但是用户当前的工作目录有可能还包括暂存区, 也许就会变得像一团乱麻。为了避免上述问题, 用户可以创建一个临时的相关工作树来修复漏洞, 解决问题之后再移除它。

这是一个新兴的领域, 若希望了解与之相关的详情, 可以参考 Git 的官方文档。

## 4.7 小结

本章主要介绍了如何更好地管理工作区和暂存区的内容，为用户创建新的提交打下了良好的基础。我们学习了如何撤销最近的提交记录，如何丢弃工作区的变更，如何追溯正在工作的分支上的变更，以及 `git reset` 命令的其他用法。现在我们已经了解了至少 3 种 `reset` 的形式。

我们还学习了如何搜索和浏览工作区、暂存区和已提交变更记录，同时还介绍了如何使用 Git 从工作区、索引以及 HEAD 指向的工作区或索引中拷贝不同版本的文件。我们还可以使用 Git 清理（删除）未跟踪文件。

本章介绍了配置工作区处理文件类型的方式，Git 系统配置忽略文件（让它们自动不被系统跟踪）的方法以及原因。

本章还介绍了如何处理不同操作系统之间的换行符差异；接着介绍了如何启用关键字表达式，如何处理二进制文件，查看和浏览特定类型的文件差异和合并记录。

我们还介绍了在工作中遇到紧急问题时使用变更暂存来应对，同时也可以方便用户在创建提交之前交互式地处理提交记录。本章介绍了如何更好地管理暂存记录，使得用户可以对其内置的操作进行功能扩展。

本章和第 3 章向读者介绍了如何为软件项目做贡献，第 2 章向读者介绍了如何管理项目版本库的克隆。

下一章将会介绍如何与他人协作，如何提交自己的工作成果，以及如何合并他人提交的变更。



## 第 5 章

# Git 协作开发

前面的第 3 章和第 4 章已经向读者介绍过如何为项目提交工作成果，但是这些内容仅限于用户自己的项目版本库克隆。第 3 章向读者介绍了如何提交新的修订，第 4 章向读者介绍了 Git 是如何帮助用户创建提交的。

本章将会介绍多种协作方式，让读者对中心式和分布式工作流有一个大致的了解。本章重点将会放在版本库级的协作开发上，同时相关的分支应用将会在第 6 章详细介绍。

本章将会向读者介绍多种协作工作流，以及它们各自的优缺点；同时读者还会了解信任链的概念，签名标签、签名合并、签名提交的使用方法。

本章的主要内容包括以下几个部分：

- 中心式和分布式工作流，裸版本库。
- 远程版本库和一次性单点协作管理。
- 推送、拉取请求以及交换补丁。
- 版本的编址——信任链。
- 使用 bundle 进行离线传输（sneakernet）。
- 标签化，轻量级标签和签名标签。
- 签名标签、签名合并和签名提交。

### 5.1 协作工作流

在使用版本控制系统时存在不同的应用层次。历史记录分析可能是唯一让人感觉有趣

的应用。第 2 章的内容就是专门讲述这一主题的。当然，查看项目历史是开发工作中非常重要的一环。

版本控制系统也经常用于某个用户使用单个机器的独立开发。第 3 章和第 4 章专门介绍了它们的具体应用。不过个人的开发工作通常也是团队协作开发工作的一部分。

版本控制系统的主要用途之一就是帮助项目团队成员进行协作开发。版本控制机制是高效同步开发某个软件的解决方案，它使得团队成员能够有效避免软件变更冲突，并帮助人们高效合并程序变更。

在进行软件项目开发时，项目或多或少都存在其他开发人员。例如其他开发成员或者维护项目人员；又或者项目非常庞大，需要子项目维护人员。其中的成员有可能和软件开发团队关系密切，也有可能只是外部辅助人员，希望能够方便地提交合适的变更（例如 bug 修复，或者帮助修正文档中的排版错误等）。现在有多种工作流可以应对上述情况：

- 中心式工作流。
- 对等网络工作流。
- 维护者工作流。
- 分层式工作流。

### 5.1.1 空版本库

有两种版本库：一种是普通的带工作目录和暂存区的非裸库，另外一种是没有工作目录的裸版本库。前一种适合个人独立开发创建修订记录，后一种适合团队协作和同步开发。

一般来说，裸版本库使用 .git 扩展名，例如 project.git；而非裸库则没有使用上述扩展名，例如 project（管理区和本地版本库中的 project/.git 文件），用户在远程版本库上执行克隆、推送和拉取操作时可以忽略该扩展名。使用 `http://git.example.com/project.git` 或者 `http://git.example.com/project` 等版本库，URL 地址的效果是一样的。

为了创建一个裸版本库，用户在执行初始化或者克隆操作时需要添加 `--bare` 选项：

```
$ git init --bare project.git
```

```
Initialized empty Git repository in /home/user/project.git/
```

## 5.1.2 和其他版本库交互

在创建了一组修订和项目的历史记录后，用户通常需要把它们和团队其他成员共享。用户需要和其他开发人员的版本库实例同步，发布自己的变更，并且从其他人员那里获取变更。从本地版本库实例的角度来说，即用户自己的远程版本库克隆，用户需要向其他版本库推送自己的变更（其中包括远程版本库或者用户自己的公共版本库），同时从其他版本库（通常是用户克隆的源版本库）中获取变更记录。在获得其他人的变更之后，用户有时需要将它们与自己的工作集成到一起，即将两个工作分支合并（或者变基），用户还可以在 pull 操作中一并完成上述工作。

通常用户并不希望公开自己的本地版本库，因为这类版本库中包含不少私人信息（其中还包括一些不成熟的想法或者工作成果）。这意味着用户完成工作后还必须添加一个额外的步骤才能使用 `git push` 命令发布自己的变更。图 5-1 演示了创建和发布提交的主要步骤，它是第 3 章类似内容的扩展版。箭头显示了 Git 执行内容拷贝的过程，其中还包括远程版本库的操作。

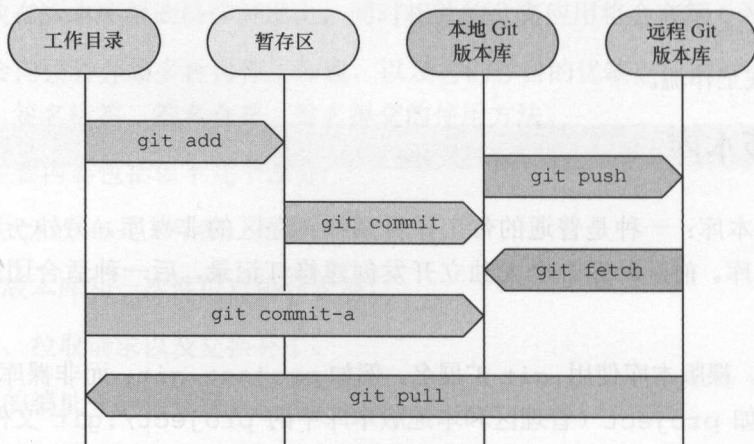


图 5-1 创建和发布提交

## 5.1.3 中心式工作流

通过分布式版本控制系统，用户采用的协作模型或多或少地都带有部分分布式特性。如图 5-2 所示，在中心式工作流中有一个中心集线器，通常是裸版本库，方便团队成员同步他们的工作成果：

每个开发人员都有其自己的中心版本库的非裸版本库克隆，它们主要用来为软件添

加新的修订。当准备好变更之后，开发人员将它们推送到中心版本库上，同时从中心共享版本库上获取其他开发人员提交的变更，因此集成工作是分布式的。该工作流可以参见图 5-2。中心式工作流的优点和缺点如下：

- 优点是配置简单，对于熟悉分布式版本控制系统和集中管理的人来说上手容易，而且支持中心式访问控制和备份功能。其对于私人的小型项目不失为一种很好的配置方案。
- 缺点是共享版本库存在单点故障问题（如果中心版本库出现问题，那么团队所有成员就无法同步变更），而且每个开发人员在发布自己的变更（将它共享给其他人）之前都必须先更新自己的版本库，并且需要合并其他人提交的变更。在这一配置中，用户还需要信任其他开发人员访问共享版本库。

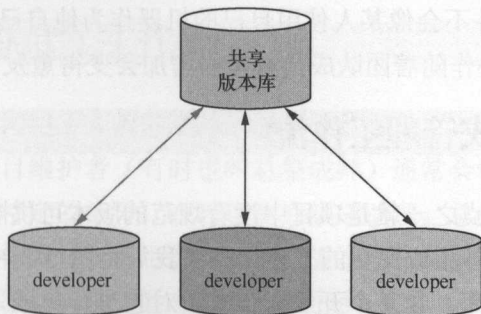


图 5-2 中心式工作流。共享版本库是裸版本库。其中连接的线代表版本库的数据流向，例如最左侧的线表示该命令是由左一版本库的所有者发送的

#### 5.1.4 对等网络或者分支工作流

和中心式工作流相反的是对等网络或者分支工作流。它没有使用单一的共享版本库，每个开发人员在私人的工作版本库（带工作目录）之外都拥有一个公共版本库，如图 5-3 所示。

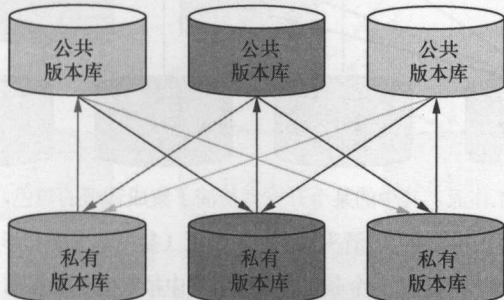


图 5-3 对等网络（分支）工作流，每个开发人员都有其自己的非裸版本库和公共裸版本库。连接的线代表发送请求的主体（发送指令的用户），向上的箭头代表推送，向下的箭头代表拉取



当准备好变更记录后，开发者将它们推送到自己的公共版本库上。为了从其他开发人员那里集成变更，用户需要从其他开发者的公共版本库上拉取变更记录。对等网络（分支）工作流的优缺点如下：

- 对等网络工作流的优点之一是集成变更记录时不需要依赖中心版本库，它是一种非常纯粹的分布式工作流。另外一个优点是，如果用户希望发布自己的变更，不必强制集成他人的变更记录，可以稍后根据自己的时间安排进行集成。这对于不需要太多配置的团队来说是一种良好的解决方案。
- 它的缺点是缺乏一个规范版本，没有集中管理机制，而且事实上，基于这种形式的工作流，用户需要和多个版本库打交道（当然这里使用 `git remote update` 命令就可以搞定）。环境配置需要开发者的公共版本库可以被团队其他成员的工作站访问，这可能并不会像某人使用自己的机器作为他自己的公共版本库那样简单，如图 5-3 所示，协作随着团队成员数量的增加会变得愈发复杂。

### 5.1.5 维护者和集成管理工作流

对等网络工作流的缺点之一就是项目中没有规范版本可供非开发人员使用，此外每个开发人员都必须亲自处理集成变更的工作。如果我们将图 5-3 中的某个公共版本库指定为规范的（官方的）版本库，让某个开发人员负责对它进行集成，那么我们就实现了集成管理工作流了（维护者工作流）。图 5-4 演示了这种工作流模型，其中裸版本库在上面，非裸版本库在下面。

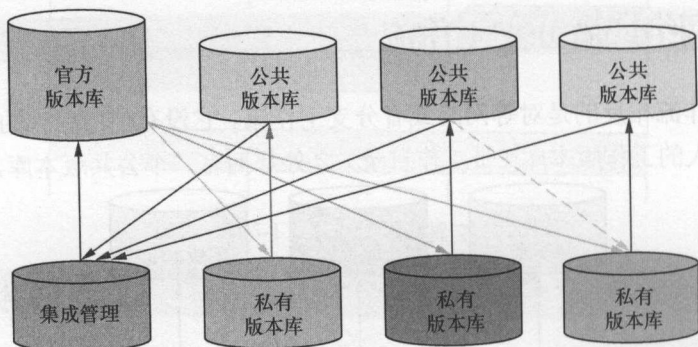


图 5-4 集成管理（维护者）工作流。其中的某个开发者扮演了集成管理的角色，其公共版本库被“推举”为项目的官方版本。指向该版本库相同颜色的箭头代表拉取变更（参见本书配套彩图，请到异步社区下载），从它指向外部的箭头代表推送。虚线代表从一个非官方的版本库中拉取变更的操作（例如一个小型的开发团队）

在这种工作流中，当准备好变更记录后，开发者将它们推送到其自己的公共版本库上，

然后告知负责维护集成的人（例如一个 pull 请求）。维护集成管理者从其他开发人员的公共版本库中拉取变更，然后将它们集成到自己的版本库中。集成管理者将上述变更合并之后，将它们推送到大家“推举”的公共版本库中。该工作流的优缺点如下：

- 优点是项目有了一个官方版本，开发人员可以不必为集成变更等待，能够安心工作，因为维护者能够随时将他们提交的变更集成。对于大型的开源项目团队来说，这是一种非常好的工作流。实际上被推荐的版本库都是通过社交互动确定的，因此可以很方便地指定其他的维护人员，这些人可以是临时性（例如时间过期）的，也可以是永久性的（例如项目分支）。
- 缺点是对于大型项目和大型团队来说，维护者集成变更的效率是个短板。为此，对于大型团队，例如 Linux 内核开发团队，最好使用层级式工作流。

### 5.1.6 层级式（主从式）工作流

层级式工作流是对等网络工作流的变体，通常被包含几百人的共同协作的大型项目采用。在这种工作流中，项目维护者（有时也叫总集成师）通常会领导若干集成经理，集成经理会管理版本库的某个特定的部分，有时也称为集成助理。总集成师的公共版本库可以通过所有集成助理的推送作为包含引用的“推举”版本库。集成助理从开发者那里获取变更，总集成师从集成助理那里获取变更，如图 5-5 所示。

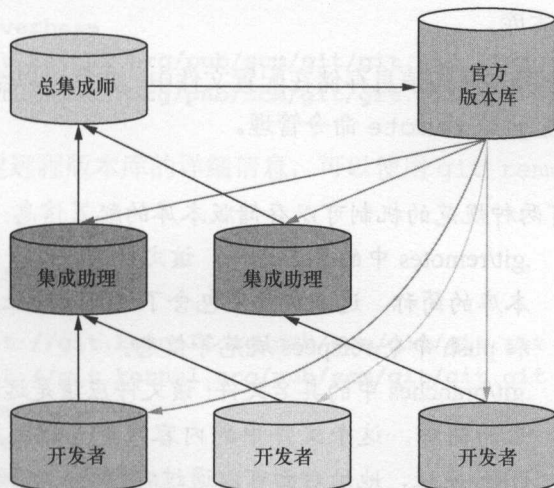


图 5-5 层级式工作流。这种项目中有一名总集成师（他的公共版本库是大家“推举”的项目官方版本），项目子系统集成管理人员叫集成助理。虚线连接的版本库实际上是开发人员和集成助手组成的配对。它们之间发起会话的人是由连接的线表示的

在层级式工作流中还存在着层级式的版本库网络。在开始工作之前，无论是开发还是合并变更，用户首先需要从项目的官方版本库拉取更新。开发者在他们的私人版本库中准备好变更记录后，会将这些变更发送给相关的子系统变更集成助手。变更记录既可以通过电子邮件发送的补丁，也可以将它们推送到开发者的公共版本库上，然后集成助手发送一个拉取请求。

变更集成助手会将上述变更合并到其负责的区域。总集成师从变更集成助手那里拉取上述变更（有时也直接从开发人员那里获取变更），同时也会兼顾软件预览发布的职责（例如为预览版软件添加标签）。该工作流的优缺点如下：

- 该工作流的优点是允许项目负责人（总集成师）代理大量的变更集成工作。这对于超大型项目或者高度分层的环境来说是非常有用的（从开发者和变更数量方面来看）。这类工作流常用来开发 Linux 内核。
- 该工作流的缺点是配置复杂。其复杂度往往超过一般的普通项目。

## 5.2 远程版本库管理

当使用 Git 进行项目协作管理时，用户经常需要访问若干其他版本库，例如在集成管理工作流中，用户至少需要访问项目成员一致“推举”的官方版本库。大多数情况下，用户需要访问多个远程版本库。

Git 允许用户将远程版本库的信息存储在配置文件中，并且可以给它指定一个别名（简称）。这类信息可以通过 `git remote` 命令管理。



有两种规范的机制可以存储版本库的配置信息：

- `.git/remotes` 中的具名文件：该文件名应该是远程版本库的简称。这个文件中包含了 URL 地址，`fetch` 和 `push` 命令 `refspecs` 规范等信息。
- `.git/branches` 中的具名文件：该文件应该是远程版本库的简称。这个文件中的内容只包含该版本库的 URL 地址，地址后面可以通过 `#` 附加分支名称。

不过时下流行的版本库中都很少采用上述两种机制，详情可以参考帮助手册的 `git-fetch(1)` 章节。



## 5.2.1 原生的远程版本库

在克隆版本库时，Git 将会为用户创建一个远程版本库，即原生的远程版本库，其中会存储用户克隆源版本库的信息，即拷贝版本库的元数据信息。用户可以使用该远程版本库获取更新信息。

这也是默认的远程版本库，例如在执行 `git fetch` 命令时，如果命令后面没有指定远程版本库的名称，那么上述命令会默认选取该原生版本库作为目标地址，除非系统对远程版本库做了特别设置，或者当前分支（分支.<分支名>.远程版本库）的配置另有规定的除外。

## 5.2.2 浏览远程版本库

为了获取目前配置的远程版本库的信息，用户可以执行 `git remote` 命令，它会列出用户拥有的所有远程版本库的别名。在一个克隆版本库中，用户至少拥有一个远程版本库——`origin`。

```
$ git remote
origin
```

为了查看和远程版本库相关的 URL 地址，可以使用 `-v / --verbose` 选项：

```
$ git remote --verbose
origin git://git.kernel.org/pub/scm/git/git.git (fetch)
origin git://git.kernel.org/pub/scm/git/git.git (push)
```

如果希望查看特定远程版本库的详细信息，可以使用 `git remote show <remote>` 子命令：

```
$ git remote show origin
* remote origin
Fetch URL: git://git.kernel.org/pub/scm/git/git.git
Push URL: git://git.kernel.org/pub/scm/git/git.git
HEAD branch: master
Remote branches:
  maint tracked
  master tracked
  next tracked
  pu tracked
  todo tracked
Local branch configured for 'git pull':
```



```

master merges with remote master
Local ref configured for 'git push':
master pushes to master (up-to-date)

```

Git 将会查找远程版本库配置、分支配置和远程版本库（为了及时更新状态）。如果用户喜欢略去访问远程版本库的步骤使用本地的缓存信息，那么可以在使用 `git remote` 命令时添加 `-n` 选项。

因为和远程版本库有关的信息是存储在版本库配置文件中的，因此用户可以在 `.git/config` 中查看它们：

```

[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = git://git.kernel.org/pub/scm/git/git.git

```

本地分支和远程分支之间（还有远程跟踪分支，远程分支的本地表示）的差异将会在第6章介绍 `refspecs` 时结合上述示例中的 `+refs/heads/*:refs/remotes/origin/*` 一起解释。

## 5.2.3 新建远程版本库

为了新建一个 Git 的远程版本库，并给它添加一个别名以便记忆，可以执行 `git remote add <shortname> <URL>` 命令：

```
$ git remote add alice https://git.company.com/alice/random.git
```

新建远程版本库的过程中，系统并不会自动拉取远程版本库的信息到本地，为了达到上述目的，用户需要在执行上述命令时添加 `-f` 选项（或者执行 `run git fetch <别名>` 命令）。

这个命令的部分参数选项会对 Git 新建远程版本库产生影响。用户可以使用 `-t <branch>` 选项在远程版本库中指定特定分支，还可以使用 `-m <branch>` 选项为远程版本库指定默认分支（以及相关的远程分支名称），也可以使用 `--mirror=push` 或者 `--mirror=fetch` 选项配置远程版本库镜像方便协作。

例如执行如下命令：

```
$ git remote add -t master -t next -t maint github \ https://github.com/
jnareb/git.git
```

将会产生下列远程版本库配置信息：

```
[remote "github"]
  url = https://github.com/jnareb/git.git
  fetch = +refs/heads/master:refs/remotes/github/master
  fetch = +refs/heads/next:refs/remotes/github/next
  fetch = +refs/heads/maint:refs/remotes/github/maint
```

## 5.2.4 远程版本库信息更新

远程版本库的信息分别存放在下面几个地方：远程版本库配置：`remote.<远程版本库名称>`，远程跟踪分支，远程首部（`refs/remotes/<remote name>/HEAD` 是一个用来表示默认远程跟踪分支的符号引用（`symref`），即远程跟踪分支的<远程版本库名称>用来指代分支），此外还有可选的单个分支配置信息：`branch.<分支名>`。

用户可以通过编辑相应的配置文件或者使用类似 `git config` 和 `git symbolic-ref` 的命令操作这些配置信息，而且 Git 还提供了多个 `git remote` 的子命令来实现上述目的。

### 远程版本库重命名

远程版本库重命名，即修改它的别名，是一项非常复杂的操作。执行 `git remote rename <old> <new>` 命令后，不仅会修改 `remote.<old>` 中的名字，而且还会修改远程跟踪分支和相关的 `refspec`，即引用日志（详情可以参考帮助手册中 `core.logAllRefUpdates` 配置变量的介绍）以及它们对应的分支配置。

### 远程版本库 URL 修改

用户可以使用 `git remote set-url` 命令为远程版本库添加或者替换 URL 地址，不过也可以直接编辑相关的配置文件进行修改。

当然用户还可以使用 `insteadOf`（还有 `pushInsteadOf`）配置变量进行替换。当用户希望使用另外一个临时的服务端主机时，上述特性是非常有用的，例如用户希望从官方网站获取 Git 软件的更新，当 `https://www.kernel.org/` 这个地址暂时无法访问时，用户就可以访问 GitHub 上的备用服务器。用户可以在配置文件中添加如下配置：

```
[url "https://github.com/git/git.git"]
  insteadOf = git://git.kernel.org/pub/scm/git/git.git
```

该特性的另外一个用处就是处理版本库迁移，用户可以使用 `insteadOf` 变量重写单用户配置文件 `~/.gitconfig`（或者 `~/.config/git/config`）中的相关配置，因而无须

手动修改每个版本库 `.git/config` 文件中的 URL。在出现多个匹配的情况下，会选用匹配数量最多的那个。



### 远程版本库的多个 URL 地址

用户可以为远程版本库设置多个 URL 地址。Git 在拉取远程版本更新时会依次尝试上述地址，并使用第一个成功访问的 URL 作为服务源，同时在推送更新时，Git 会同步推送更新到上述所有 URL 地址的服务主机上。

## 远程版本库跟踪分支列表编辑

修改 URL 常见的一种情况是修改远程版本库的跟踪分支列表（拉取内容的列表）：用户可以使用 `git remote set-branches` 命令（通过一款当前最新的高效 Git 软件实现）或者直接手动编辑配置文件实现该目的。

需要注意的是，从远程版本库跟踪列表中移除一个分支并没有删除该跟踪分支，后者只是不再更新了。

## 远程版本库的默认分支设置

远程版本库上不一定必须有一个默认分支，不过它可以让用户通过远程版本库名称替代某些特定的远程跟踪分支（例如 `origin/master`）。这类信息一般存放在符号化引用 `<远程版本库名称>/HEAD` 中（例如 `origin/HEAD`）。

用户可以使用 `git remote set-head` 命令设置它，`--auto` 选项的作用是选择远程版本库中的当前分支：

```
$ git remote set-head origin master
$ git branch -r
origin/HEAD -> origin/master
origin/master
```

用户还可以使用 `--delete` 选项删除远程版本库上的默认分支。

## 远程版本库跟踪分支删除

当远程版本库上的一个公共分支被删除后，Git 仍然会保留相关的远程跟踪分支。这样做的原因是用户可能已经基于上述分支添加了新的变更。当然，用户也可以使用 `git`



`branch -r -d` 命令删除远程跟踪分支，或者可以通过 `git remote prune` 命令让 Git 系统对远程版本库中的远程跟踪分支进行精简，它的作用和 `git fetch` 命令搭配 `--prune` 选项是类似的，当然前提是用户必须设置好 `fetch.prune` 和 `remote.<name>` 等精简配置变量。用户可以使用 `git remote prune` 命令搭配 `--dry-run` 选项或者使用 `git remote show` 命令检查远程跟踪分支中哪些是需要被清理的内容。

用户还可以使用 `git remote delete` 命令（或者 `git remote rm` 命令）将整个远程版本库删除。同时系统还会将与之相关的远程跟踪分支移除。

### 5.2.5 兼容不规则 workflow

在很多协作 workflow 中，例如维护者（集成管理）workflow，用户需要从一个 URL 地址获取更新（从大家“推举”的公共版本库），但是还需要将自己的更新推送到另外一个 URL（自己的公共版本库）。如图 5-4 所示，开发者需要和其他 3 个版本库交互——其需要从“推举”版本库（浅红色的）获取更新并集成到自己的私人版本库（深色的），同时还需要推送自己的工作成果到开发者公共版本库（浅色的）上。

在这种不规则 workflow 中（3 个版本库），用户拉取或者推送的远程版本库通常是默认的 `origin` 远程版本库（或者 `remote.default`）。一个备选的版本库配置方案是将用户推送的目标版本库当作一个独立的远程版本库，当然有可能用户还需要使用 `remote.push Default` 变量将其设置为默认主机。

```
[remote "origin"]
    url = https://git.company.com/project
    fetch = +refs/heads/*:refs/remotes/origin/*
[remote "myown"]
    url = git@work.company.com:user/project
    fetch = +refs/heads/*:refs/remotes/myown/*
[remote]
    pushdefault = myown
```

用户还可以在单个分支配置中将它设置为推送远程主机（`pushremote`）：

```
[branch "master"]
    remote = origin
    pushremote = myown
    merge = refs/heads/master
```

另外一个备选方案是使用单一的远程版本库（甚至只有 `origin`），但是分别设定独立



的推送 URL 地址。当然，这个方案有一个小小的瑕疵，即用户不能为版本库设置独立的远程跟踪分支（因为@{upstream}之外不存在@{push}这样的修饰符来表示相应的特定远程跟踪分支的快捷方式，不过 Git 2.5.0 之后的软件版本已经支持上述语法了）：

```
[remote "origin"]
  url = https://git.company.com/project
  pushurl = git@work.company.com:user/project
  fetch = +refs/heads/*:refs/remotes/origin/*
```

## 5.3 传输协议

一般来说，远程版本库配置文件中的 URL 都包含传输协议信息，即远程主机的地址和版本库的路径。有些时候，远程版本库服务端为用户提供了使用多种协议访问它的支持，用户可以根据喜好选择其中的一种。本节将会介绍可供用户选择的多种传输协议。

### 5.3.1 本地传输

如果其他版本库是放在本地的其他文件系统上的，那么用户可以使用下列语法格式指定访问该版本库的 URL 地址：

```
/path/to/repo.git/
file:///path/to/repo.git/
```

前一种实现了 Git 克隆版本库操作中--local 选项的功能，它绕过了 Git 的智能感知机制，只是简单地执行了拷贝操作（或者是.git/对象下的所有文件链接，当然用户还可以使用--no-hardlinks 选项禁用此功能），后者虽然效率比较低，但是可以获取一个干净的版本库拷贝。

这是一种非常好的办法，它可以快速地从某人正在工作的版本库中获得工作成果，也可以通过授予文件系统一定访问权限和他人共享自己的工作成果。

有一种特殊的情况，单个"." 符号可以代表当前版本库，下列命令的含义

```
$ git pull . next
```

是和下文等效的：

```
$ git merge next
```

### 哑传输协议

某些传输过程不需要任何 Git 智能感知服务——它们不需要在服务端安装 Git 软件（对于智能传输过程至少需要用到 `git-upload-pack` 或者 `git-receive-pack` 组件），不过其中大部分需要通过 `git update-server-info` 根据版本库引用、对象和打包文件（以某种方式拷贝）生成的附加信息才能工作。

### Rsync 传输协议：不安全

作为 Git 最初支持的最古老传输协议之一，通过它用户可以从远程版本库上执行拉取、推送、读取和写入等操作，即 `rsync` 协议，它的 URL 类型如下所示：

`rsync://host.example.com/path/to/repo.git/`  
`rsync` 协议已经过时了，因为它无法保证在获取数据时得到合适的序列，如果用户从一个非静态版本库中拉取数据，那么可能会得到无效的数据。另外，它的传输速度很快并且支持断点续传。当然，如果用户通过一个不太稳定的网络环境执行初始化克隆操作时出现问题，那么最好采用 `bundle` 替代 `rsync` 协议，相关内容将会在后续章节介绍。



### FTP(S)和哑 HTTP(S)传输协议：效率低

这类传输只需要提供相应的服务器即可（例如 FTP 服务器或者 Web 服务器），然后通过 `git update-server-info` 命令进行数据更新。当从这样一台服务器上拉取数据时，Git 会通过名为 `commit walker` 的下载器下载相应的分支和标签数据，遍历整个提交链条，下载对象、包含新修订的压缩包以及其他数据（例如修订的文件内容）。

这种传输是非常低效的（由于带宽的原因，但是主要还是由于网络延迟），但是另外它还支持断点续传。尽管如此，还有不少优于哑传输协议的解决方案，当网络环境不稳定以致于用户无法获取版本库克隆时，用户可以试试借助 `bundle` 软件（详情可参考 5.3.3“使用 `bundle`”）。



进行离线传输”的内容)。

向一个哑服务器推送内容只能通过 HTTP 和 HTTPS 协议,同时需要 Web 服务器支持 WebDAV 功能,而且采用的 Git 软件必须使用外部的链接库进行编译生成。FTP 和 FTPS 协议是只读的(即只支持克隆、下载和拉取操作)。

### 5.3.2 智能传输

当用户希望获取的版本库在另外一台机器上时,就需要借助 Git 服务器才能访问了。当前常见的设备是 Git 智能感知服务器。智能下载协议可以自动选择必要的软件版本,然后创建一个自定义打包文件发送到客户端。同时,在推送变更过程中,Git 服务端会和用户机器上的 Git 软件(客户端)交互,以便确定哪些修订版本需要上传。

Git 智能感知服务器会使用 `git upload-pack` 命令下载更新,使用 `git receive-pack` 命令推送变更。如果无法在 `PATH` 环境变量(也可能在安装主目录下)中找到相应的地址,用户可以通过 `--upload-pack` 和 `--receive-pack` 选项告诉 Git 系统获取更新和推送变更的目标地址,当然也可以使用远程版本库配置变量 `uploadpack` 和 `receivepack` 进行设置。

极个别情况下(例如版本库使用的子模块是通过一个版本较低的 Git 软件进行访问的,以致于该软件无法有效识别其他模块),Git 传输过程是向下兼容的,即客户端和服务端可以共同协商,采用二者都支持的协议进行数据传输。

#### Git 原生通信协议

原生的传输协议采用的格式为 `git://URLs`,它支持只读式的匿名访问(从理论上说,用户可以通过命令行或者布尔值配置变量 `daemon.receivePack` 让 Git 的 `receive-pack` 服务支持推送,不过并不建议用户使用这种机制,即使在一个完全封闭的本地网络中也是如此)。

Git 原生通信协议是没有授权验证机制的,其中还包括服务端授权验证,因此在不安全的网络环境中要慎用。上述协议 Git 的 TCP 守护进程一般会监听 9418 端口,因此用户必须有权限能够访问该端口(穿透防火墙)才能使用该协议。



## SSH 协议

安全外壳传输协议 (SSH) 提供了支持安全验证的读写访问功能。Git 在服务端执行 `git upload-pack` 或者 `git receive-pack` 命令时就是采用的 SSH 协议进行远程调用的。该协议不支持匿名和未授权访问, 因此用户可以专门为它创建一个 `guest` 账户 (使用简单密码或者空密码)。

采用公私钥授权之后, 用户每次连接时无需输入密码, 只需第一次连接时提供密码即可——解密受保护的私钥。详情可以参考后续身份验证章节的内容。

在使用 SSH 协议时, 用户可以将 `ssh://` 作为 URL 语法的一部分:

```
ssh://[user@]host.example.com[:port]/path/to/repo.git/
```

当然也可以使用类 `scp` 的语法:

```
[user@]host.example.com:path/to/repo.git/
```

此外 SSH 协议还支持 `~username` 表达式, 其在语法形式上和原生的 Git 传输协议类似 (`~` 代表当前登录用户的主目录, `~user` 代表当前主目录的用户):

```
ssh://[user@]host.example.com[:port]/~[user]/path/to/repo.git/
[user@]host.example.com:~[user]/path/to/repo.git/
```

SSH 在服务端采用首次用户验证机制: 它会记住用户上一次登录时提供的密钥, 并在此次用户登录时向用户发出警示信息, 确认该信息是否有误 (服务端的密钥有可能已经发生变更, 例如重新安装 SSH 服务器软件)。用户可以在首次连接服务器时对密钥指纹进行校验。

## 智能 HTTP(S) 协议

Git 也支持智能 HTTP (S) 协议, 不过它需要借助 Git 感知 CGI 和服务模块, 例如 `git-http-backend` (它本身就是一个 CGI 模块)。由于设计特性, Git 可以自动将简单传输 URL 转换成智能 URL 地址。

与此相反, Git 感知 HTTP 服务可以向下兼容哑传输协议 (至少支持拉取操作, 不过它不支持基于 WebDAV 的哑 HTTP 推送)。这个特性允许用户使用相同的 HTTP (S) 的 URL 地址进行哑传输和智能传输访问:

```
http[s]://[user@]host.example.com[:port]/path/to/repo.git/
```



默认情况下, 如果不做任何配置, Git 还支持匿名下载 (`git fetch`、`git pull`、`git clone` 和 `git ls-remote`), 不过在上传信息时, 需要客户端进行授权验证(`git push`)。

如果访问某个版本库时需要授权验证, 那么一般会通过 HTTP 服务端软件进行标准的 HTTP 验证。如果需要发送密码, 最好采用 SSL/TLS 协议进行 HTTPS 加密传输, 并且服务端的凭据是经过验证的 (使用服务端 CA 证书)。

### 5.3.3 使用 bundle 进行离线传输

有时用户的机器无法直接访问 Git 版本库所在主机, 又或者服务端主机关闭了, 但是用户又希望将某些变更拷贝到其他机器上。也有可能用户的网络出现故障。有时也许是因为安全因素用户无法访问本地网络。有时也许是因为无线/以太网卡失灵。

这时 `git bundle` 命令就能派上用场了。该命令会将本该通过无线网络进行传输的对象和引用打包成名为 `bundle` 的特定二进制文件 (例如只包含分支的打包文件等)。此时用户需要声明哪些提交对象应该被打包 (通常在进行网络传输时网络协议会自动为用户处理)。



当用户采用了某个智能协议之后, 上述步骤通常在网络协商过程发生, 客户端会告知服务端自己本地版本库中包含了哪些内容, 希望从服务端版本获取哪些引用, 并找到相应的修订版本。然后服务端通过客户端发送的信息创建一个打包文件, 返回客户端的只是一些必要文件, 从而减少带宽占用。

接下来, 由于某些原因, 用户希望将它们移动到自己的机器上 (例如 `sneakernet`, 这意味着需要将 `bundle` 文件拷贝到移动硬盘上来达到文件共享的目的)。用户可以通过 `git clone` 和 `git fetch` 命令并用 `bundle` 文件名替换版本库 URL 进行相关操作。



#### Git 传输代理

有时用户无法直接访问服务器, 例如在安装了防火墙的局域网中, 因此用户可能需要借助代理访问外网。

对于原生的 Git 通信协议 (`git://`), 用户可以使用配置变量 `core.gitProxy` 或者环境变量 `GIT_PROXY_COMMAND` 声明一个代理命令, 例如 `ssh`。



可以使用 `core.gitProxy` 值——`<command> for <remote>` 这样特殊的语法对单个远程版本库进行配置。用户可以通过配置变量 `http.proxy` 或者环境变量 `curl` 声明 HTTP 代理服务器, 以便调用 HTTP(s) 协议 (`http(s)://`)。用户可以在单个远程版本库中使用配置变量 `remote.<remote name>.proxy` 进行相关配置。

用户可以配置 SSH (使用它的配置文件, 例如 `~/.ssh/config`) 来使用隧道 (端口转发) 或者使用一个代理命令 (例如 `netcat/nc` 或者 `ssh` 的 `netcat` 模式)。SSH 代理是一个非常棒的解决方案。如果既没有隧道也没有代理可用的话, 还可以使用 `ext::` 通信助手, 本章后续部分会对它进行详细介绍。

## 使用 bundle 克隆和更新

假定用户希望把项目历史记录从 A 机器 (例如用户的工作电脑) 上转移到 B 机器 (例如服务器主机) 上, 但是上述两台电脑之间不能直连。

首先, 我们创建一个包含整个 `master` 分支历史记录 `bundle` 文件, 然后给这些历史记录添加一个标签以方便记忆, 接下来执行如下操作:

```
user@machineA ~$ cd repo
user@machineA repo$ git bundle create ../repo.bundle master
user@machineA repo$ git tag -f lastbundle master
```

这里的 `bundle` 文件是在工作目录之外创建的。这是一个值得考虑的因素, 存放在版本库外部意味着用户无须担心因为误操作而将它添加到项目历史记录中, 或者不得不添加一条新的忽略规则。`*.bundle` 文件扩展名也是约定俗成的。



基于安全因素考虑, 为了避免泄漏没有清理干净的项目历史信息 (例如偶尔将带密码信息的文件提交到历史记录中), Git 只允许从兼容 `git show-ref-compatible` 命令的引用和分支、远程跟踪分支和标



签中获取更新数据。

相同的限制因素也适用于创建 bundle 文件。这意味着上述示例中(由于实现的原因)用户不能执行 `git bundle create master^1` 这样的命令。当然,因为可以控制服务端,用户为 `master^` 创建一个新分支,回退到 `master` 节点或者在 `master^` 签出脱离首部等,都是不错的解决方案。

接下来用户就可以把刚才创建的 `repo.bundle` 文件转移到 B 机器上了(通过电子邮件、U 盘或者 CD 光盘等)。因为该 bundle 文件结构是自包含的,是项目历史记录完整子集,即包含了根提交以下的所有节点,用户可以通过克隆操作创建一个新的版本库,只需将版本库 URL 替换成 bundle 文件名即可:

```
user@machineB ~$ git clone repo.bundle repo
Initialized empty Git repository in /home/user/repo/.git/
warning: remote HEAD refers to non-existent ref, unable to checkout.
```

```
user@machineB ~$ cd repo
user@machineB repo$ git branch -a
remotes/origin/master
```

幸好我们没有绑定 HEAD, 因此 `git clone` 命令执行后无法确认当前分支, 这样可以方便用户签出。

```
user@machineB repo$ git bundle list-heads ../repo.bundle
5d2584867fe4e94ab7d211a206bc0bc3804d37a9 refs/heads/master
```



因为 bundle 文件可以被当作一个远程版本库, 这里我们可以使用 `git ls-remote ../repo.bundle` 替代 `git bundle list-heads ../repo.bundle`。

因此, 由于 bundle 文件自身的特性, 我们需要特别声明签出的分支(如果其中也打包了 HEAD 信息, 那么这一步骤也不是必需的):

```
user@machineB ~$ git clone repo.bundle --branch master repo
```

接下来补上签出这一步骤来解决上述问题（当然用户使用的 Git 软件版本应该尽量是最新的）：

```
user@machineB repo$ git checkout master
Branch master set up to track remote branch master from origin.
Already on 'master'
```



这里使用了 `git checkout <branch>` 命令的一个特殊用法：因为 `master` 分支并不存在，但是某个远程版本库上却有一个同名的远程跟踪分支（这里是 `origin/master`），Git 会假定用户的意图是希望为研发工作创建一个本地分支，方便用户将它发布到 `origin` 版本库的 `master` 分支。在版本比较旧的 Git 软件中，用户必须显式声明这一点：

```
user@machineB repo$ git checkout -b master
--track origin/master
```

这会定义一个名为 `origin` 的远程版本库，并使用如下配置：

```
[remote "origin"]
    url = /home/user/repo.bundle
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

为了更新 B 机器上克隆自 `bundle` 的版本库，用户在执行 `fetch` 或者 `pull` 命令之后，需要用包含增量更新的 `bundle` 文件替换原生的 `/home/user/repo.bundle`。

本示例中，为了创建一个包含最后一次传输之前变更历史的 `bundle` 文件，用户需要在 A 机器上执行如下命令：

```
user@machineA repo$ git bundle create ../repo.bundle \
    lastbundle..master
user@machineA repo$ git tag -f lastbundle master
```

上述操作会打包 `lastbundle` 标签之前的所有变更记录，该标签可以代表上述 `bundle` 文件拷贝的内容（双点符号的含义可以参考第 2 章）。在创建一个 `bundle` 文件之后，将会



更新标签（使用 `-f` 选项替换它），这就可以像首次创建 `bundle` 文件那样，这样在下次创建 `bundle` 文件时就可以根据当前的记录进行增量创建了。

然后用户只需将 `bundle` 文件拷贝到 B 机器，替换掉旧的那个文件。为此，用户只需简单地执行 `pull` 命令更新版本库即可：

```
user@machineB repo$ git pull
From /home/user/repo.bundle
   ba5807e..5d25848   master    -> origin/master
Updating ba5807e..5d25848
Fast-forward
```

## 使用 `bundle` 更新现存版本库

有时也许用户已经有一个版本库的克隆了，只是因为网络故障的原因，又或者用户在本地局域网之外的地方，目前无法访问服务器。最终的结果是，用户已经有一个版本库了，但是不能直接访问它的上游（即版本库克隆的源数据）。

假定用户不希望将整个版本库打包，如前文所述，用户需要找到在目标版本库中（用户本机）声明截止点的方法。

用户可以使用第2章介绍的知识指定希望打包到 `bundle` 文件的修订历史区间。唯一的不足是修订历史记录必须从某个分支或标签开始（任何 `git show-ref` 命令可以接受的对象参数）。当然，用户还可以使用 `git log` 命令查看上述区间中的内容。

将指定区间的修订记录打包成 `bundle` 文件常用的解决方案遵循以下步骤：

- 使用两个版本库都有的标签：

```
machineA repo$ git bundle create ../repo.bundle v0.1..master
```

- 根据提交创建时间创建一个截止点：

```
machineA repo$ git bundle create ../repo.bundle --since=1.week master
```

- 打包最近的几个提交记录创建 `bundle` 文件，通过提交记录数目限定修订区间：

```
machineA repo$ git bundle create ../repo.bundle -5 master
```



打包的数据多多益善，否则你会看到类似下面的提示：

```
user@machineB repo$ git pull ../repo.bundle
master error: Repository lacks these prerequisite
commits: error: ca3cdd6bb3fcd0c162 a690d5383bdb
8e8144b0d2
```

用户可以使用 `git bundle verify` 命令检查 bundle 文件中是否包含相关版本库所需的提交记录。

将打包数据迁移到 B 机器上之后，用户可以像使用普通的版本库那样对 bundle 文件进行一次拉取（pull）操作（将 URL 地址或者远程版本库名称替换成 bundle 文件名即可）：

```
user@machineB repo$ git pull ../repo.bundle master
From ../repo.bundle
* branch          master      -> FETCH_HEAD
Updating ba5807e..5d25848
```

如果用户不想处理上述合并，可以从远程跟踪分支上下载相关内容（这里使用了 `<remote branch>:<remote-tracking branch>` 表达式，它的用法和 `refspec` 类似，详情可以参考第 6 章）：

```
user@machineB repo$ git fetch ../repo.bundle \
refs/heads/master:refs/remotes/origin/master
From ../repo.bundle
ba5807e..5d25848 master -> origin/master
Updating ba5807e..5d25848
```

或者用户可以使用 `git remote add` 命令新建一个快捷方式，使用 bundle 文件名代替版本库的 URL 地址，然后就可以像前文所述那样对 bundle 文件进行了。

## 使用 bundle 初始化克隆

智能传输协议比哑协议传输效率更高。另外，智能传输协议支持克隆的断点续传功能的实现仍然遥遥无期（至少 Git 2.7.0 不支持该功能，也许将来某一天会支持该功能）。对于包含长远历史记录和大量项目文件的大型项目来说，初始化克隆的文件可能会非常大（例如 `linux-next` 项目文件大概有 800MB），而且会非常耗时。如果用户的网络环境不稳定，那么就有可能出现问题。

用户可以根据源版本库创建一个 bundle 文件，例如使用如下命令：

```
user@server ~$ git --git-dir=/dir/repo.git bundle create --all HEAD
```

某些服务商还提供了帮助用户初始化克隆的 bundle 文件。目前比较常见的做法(约定)是给定 URL 的版本库还有一个对应的 bundle 文件,只不过其后缀名是以 bundle 结尾的。例如 `https:// git.example.com/git/repo.git` 对应的 bundle 文件地址是 `https://git.example.com/git/repo.bundle`。

用户可以下载这样的 bundle 文件,该文件是一个普通的静态文件,并且支持任何支持断点续传功能的协议,如 HTTP(S)、FTP(S)和 rsync,甚至可以使用 BitTorrent(配合支持断点续传功能的客户端软件)。

### 5.3.4 远程版本库传输助手

当 Git 无法识别某个特定的传输协议时(内置协议不支持),它会尝试使用相应的远程版本库助手解析该协议。当 Git 服务解析某个协议时,用户会看到如下错误提示信息:

```
$ git clone ssh://git@example.com:repo
Cloning into 'repo'...
fatal: Unable to find remote helper for 'ssh'
```

上述异常信息的含义是指 Git 系统尝试查找 git-remote-ssh 处理 ssh 协议(实际上因为印刷错误,应该是 ssh),但是却无法找到与之相关的可执行程序名称。用户可以使用 `<transport>::<address>` 这样的语法,显式明确指定某个特定的远程版本库助手, `<transport>` 中定义了助手名称 (git remote-`<transport>`), `<address>` 中定义了助手需要查找的版本库地址。

时下最新版 Git 程序对哑 HTTP、HTTPS、FTP 和 FTPS 等协议都提供了远程版本库助手,它们分别是 git-remote-http、git-remote-https、git-remote-ftp 和 git-remote-ftps。

#### 远程版本库助手传输应答

Git 内置的两种常用的远程版本库助手可以用来充当智能传输代理:一种是通过双向通信或者结对管道连接远程服务器的 git-remote-fd,另外一种是使用外部命令连接远程服务器的 git-remote-ext。

对于后者,用户可以使用 `"ext::<command>[ <arguments>...]"` 这样的语法指定版本库的 URL, Git 通过执行指定的命令与服务端相连,将标准的输入命令发送到服务



端，然后返回相应的标准输出响应结果。这种数据可能会通过 `git://server`、`git-upload-pack`、`git-receive-pack` 或者 `git-upload-archive` 中的某一个进行传递（视情况而定）。


例如，假定用户的版本库在某个局域网主机上，而且可以通过 SSH 进行访问。不过出于安全性因素的考虑，该主机和互联网是隔离的，那么这时候用户就需要先通过网关主机：`login.example.com`。

```
user@home ~$ ssh user@login.example.com
user@login ~$ ssh work
user@work ~$ find . -name .git -type d -print
./repo/.git
```

现在的问题是：由于安全性方面的原因，网关主机既没有安装 Git 软件（降低被攻击的风险），也不能访问用户的版本库（它使用的文件系统有别于普通电脑）。这意味着用户无法使用普通的 SSH 协议。但是 SSH 传输可以通过 `git-receive-pack` / `git-upload-pack` 等组件通过 SSH 将版本库 URL 作为执行参数进行远程访问。因此用户可以使用 `ext::` 远程版本库助手进行访问：

```
user@home ~$ git clone \
    "ext::ssh -t user@login.example.com ssh work %S 'repo'" repo
Cloning into 'repo'...
Checking connectivity... done.
```

这里的 `%S` 将会被 Git 分别替换成执行拉取的 `git-receive-pack` 和执行推送的 `git-receive-pack` 对应的全名。如果登录内部主机采用的是交互式授权验证机制（例如输入密码），则还需要用到 `-t` 选项。需要注意的是，用户需要在最终的克隆操作中指定版本库名称（这里的名字是 `repo`），否则 Git 将会使用上述命令（`ssh`）作为版本库的名称。



用户还可以使用 `"ext::ssh [<parameters>...]"`  
`%S '<repository>'"` 为 SSH 传输声明特定的选项。  
 例如使用选择密钥对，就不需要再编辑 `.ssh/config` 文件了。

这并不是唯一的解决方案，还有不需要内置功能支持，通过代理进行 SSH 传输的方法，例如支持原生的 `Git://` 协议（结合 `core.gitProxy` 一起使用）和 HTTP 协议（结合 `http.proxy` 一起使用），当然用户还可以通过配置 `.ssh/config` 文件（参考



ProxyCommand) 或者创建一个 SSH 信道来实现。

另外一方面, 用户可以使用 `ext::` 远程版本库助手代理 `git://` 协议, 例如借助 `socat` 软件, 该软件支持多个服务器共享一个代理。详情可以参考帮助手册 `git-remote-ext(1)` 相关章节。

## 使用外源 SCM 版本库作为远程版本库

远程版本库助手机制非常强大。它可以用来和其他版本控制系统交互, 用户可以像使用原生的 Git 版本库那样使用其他类型的版本库。因为其他软件中并没有内置类似的助手功能 (除非用户对 Git 软件源代码中的 `contrib/` 区域做了统计), 用户可以通过 `git-remote-hg` (或者 `gitifyhg`) 模块访问 Mercurial 版本库, 使用 `git-remote-bzr` 模块访问 Bazaar 版本库。

安装上述模块之后, 这类远程版本库助手适配器就支持用户像在 Git 版本库中那样, 在 Mercurial 或者 Bazaar 的版本库中通过 `<helper>::<URL>` 这样的语法, 执行克隆、拉取和推送操作了。例如, 用户希望在 Mercurial 版本库中执行克隆操作, 那么可以执行如下命令:

```
$ git clone "hg::http://hg.example.com/repo"
```

如果用户不喜欢在版本库 URL 之前使用 `<helper>::` 前缀, 那么还可以使用配置变量 `remote.<remote name>.vcs`。通过上述方法, 用户可以像在 Git 中那样, 在其他原生的 VCS (版本控制系统) 中使用相同的 URL。当然, 用户需要留意不同版本库之间的差异, 以及远程版本库助手的不足之处。其中某些特性并不能完全转换或者转换的效果不佳, 例如 Git 中的章鱼合并 (有两个父提交), 以及 Mercurial 中的多个匿名分支 (首部)。使用远程版本库助手时也不能进行错误修复、使用目标原生语法替换修订引用, 以及由于版本库约定创建的冗余信息——当变更版本控制系统时也可以也应该使用一次性约定进行清理 (类似的清理工作可以借助像 `reposurgeon` 这样的第三方工具完成)。

通过远程版本库助手, 用户甚至可以处理严格意义上非版本控制的版本库, 例如通过 Git-Mediawiki 项目, 用户可以使用 Git 软件查看和编辑基于 MediaWiki 的 wiki 页面 (例如维基百科), 即将页面历史作为 Git 版本库:

```
$ git clone mediawiki::http://wiki.example.com
```

此外, 远程版本库助手还支持其他的传输协议或者存储选项, 例如 `git-remote-s3bundle` 可以在 Amazon S3 服务上将版本库存储为一个 bundle 文件。

### 5.3.5 凭据/密码管理

大部分情况下，除了本地协议之外，将变更发布到远程版本库上都需要用户身份验证（用户自身的身份验证）和 Git 系统授权（给定用户是否有权限推送变更）。有时从版本库拉取更新记录也需要身份验证。常用于身份验证的信息是用户名和密码。如果用户不介意信息泄漏，就可以将用户名添加到 HTTP 和 SSH 的版本库 URL 中，或者还可以使用验证助手机制。虽然技术上来说没什么问题，但是用户永远不应该将密码放在 URL 中，因为密码可能会被其他不怀好意的人看到，例如他们在监听进程通信时。

除了底层传输引擎的内在机制、ssh 的 SSH\_ASKPASS 变量，以及基于 curl 传输的 ~/.netrc 文件，Git 自身也提供了相关的解决方案。

#### 密码验证

某些 Git 命令支持交互式密码验证（当然还需要知道相关的用户名），例如 git svn、HTTP 接口以及 IMAP 授权（可以用来调用外部程序）。该程序可以被相应的对话框（又称为域，描述密码的用途）调用，Git 从上述程序的标准输出中读取密码。

Git 将会以此在以下几个地方验证用户的密码和用户名信息，详情可以参考帮助手册的 gitcredentials(7)。

- 程序通过环境变量 GIT\_ASKPASS 设定的参数，如果设置了该变量值（Git 声明的环境变量优先级永远比配置变量高）。
- 然后检查配置变量 core.askpass 是否做了相关设置。
- 然后检查环境变量 SSH\_ASKPASS 是否做了相关设置。
- 然后检查用户终端对话框。

“askpass”外部程序经常会检查用户桌面环境（安装后，如果有必要的话）。例如 (x11-)ssh-askpass 提供了一个纯文本的 X-Window 对话框来验证用户名和密码，此外还有 GNOME 环境的 ssh-askpass-gnome，KDE 环境的 ksshaskpass，MacOS X 下的 mac-ssh-askpass，MS Windows 下的 win-ssh-askpass 可供用户选择。Git 还提供了一款使用 Tcl/Tk-git-gui--askpass 开发的跨平台密码验证对话框来适配 git gui 图形界面和 gitk 历史记录查看器。

### Git 配置的优先级

可以通过多种方法配置 Git 中命令的行为。它们遵循如下规则：少数服从多数，先来优于后到：



- 命令行选项，例如 `--pager`；
- Git 特别声明的环境变量，例如 `GIT_PAGER` 和 `GIT_ASKPASS`（这类变量常使用 `GIT_` 前缀）；
- 配置选项（在某个配置文件中，配置选项也有相应的优先级），例如 `core.pager` 和 `core.askpass`；
- 通用环境变量，例如 `PAGER` 和 `SSH_ASKPASS`；
- 内置的默认属性，例如 `less` 打印或者终端提示。

## SSH 公钥授权

SSH 传输协议除了密码之外还有别的授权验证机制。公钥授权就是其中之一。这种机制可以避免用户一次又一次地输入密码。此外，版本库主机服务提供的 SSH 访问也可能用到它，这可能是由于一个用户进行验证时，基于其公钥不需要使用一个独立的账户（这也是 `gitolite` 的用途）。

其设计理念是用户创建公钥/私钥对一起运行，例如 `ssh-keygen` 软件。公钥会被发送到服务端，例如使用 `ssh-copy-id`（在远程版本库服务器上的 `~/.ssh/authorized_keys` 文件夹下添加名为 `*.pub` 的公钥文件）。然后用户就可以在自己的本地机器上使用私钥访问远程版本库了，例如使用 `~/.ssh/id_dsa` 这样的私钥文件。如果不是默认的验证密钥，用户还需要使用特定的验证凭据文件为给定主机进行 `ssh` 配置（Linux 下配置文件在 `~/.ssh/config` 文件夹中）。

另外一种简便的方法是使用像 `ssh-agent` 这样的授权代理软件进行公钥授权（在 Windows 上使用基于 PuTTY 的 `Pagenat` 软件）。通过代理还可以让用户更方便地使用带密码保护的私钥。对于代理来说，用户只需要在添加密钥时输入一次密码即可（当然还需要用户执行一些其他操作，例如在 `ssh-agent` 上执行 `ssh-add` 命令）。

## 验证助手

一次又一次地输入相同的凭据信息的用户体验是非常糟糕的。对于 SSH，用户可以使用公钥进行授权验证，这一点也是它有别于其他协议的地方。Git 凭据配置提供了两种办法简化问题。



第 1 种方法是对于给定的授权上下文默认用户名（如果 URL 中没有提供用户名的话）提供静态配置，例如主机名：

```
[credential "https://git.example.com"]
    username = user
```

这一点对于没有提供安全的存放凭据机制的用户来说是非常有利的。

第 2 种方法是使用 Git 可以获取用户名和密码的外部程序作为凭据助手。这些程序通常都有桌面环境或者操作系统提供安全存储机制的接口（例如 `keychain`、`keyring`、`wallet` 和 `credentials manager` 等）。

一般来说，Git 软件至少都内置了缓存和存储助手。缓存助手（`git-credential-cache`）可以临时将凭据信息存放在内存中，默认情况下它存储密码和用户名的时间是 15 分钟。存储助手（`git-credential-store`）可以将用于授权验证的信息长期存放在硬盘上，并且存放的文件只能由对应的用户读取（例如 `~/.netrc` 文件）；当然还有第三方的 `netrc` 助手（`git-credential-netrc`），它可以对 `netrc/authinfo` 中的文件进行 GPG 加密。

如前面示例所述，如果希望使用某个凭据助手的功能和选项，既可以通过全局性的，也可以通过局部验证授权上下文环境实现。全局凭据的配置如下：

```
[credential]
    helper = cache --timeout=300
```

这将创建一个寿命为 300 秒的 Git 缓存凭据。如果凭据助手的名称不是绝对路径（例如 `/usr/local/bin/git-kde-credentials-helper`），Git 将会为助手名称前面添加 `credential-` 前缀。用户如果希望查看哪些助手是可供选择的，可以使用 `git help -a | grep credential-` 命令（排除名称中包含 `--` 的名字，因为这和内部实现有关）。

现有的凭据助手一般采用的是相关桌面环境的安全存储机制。当用户使用它们时，只需要输入一次密码将它们解锁即可（某些助手可以在 Git 源码中的 `contrib/` 区域找到）。现有针对 Gnome Keyring 的 `git-credential-gnome-keyring` 和 `git-credential-gnomekeyring`，针对 MacOS X Keychain 的 `git-credential-osxkeychain`，针对 MS Windows 的 `git-credential-wincred` 和 `git-credential-winstore` 的凭据管理和存储助手组件。

Git 将会采用凭据配置中最常用的授权验证上下文环境，因此如果用户希望通过路径名辨别 HTTP 的 URL（例如在同一主机不同版本库分别提供各自不同的用户名），那么需要将配置变量 `useHttpPath` 的值设置为 `true`。如果配置了多种上下文凭据助手，Git 将会依



次调用它们，直到系统获得匹配的用户名和密码。



在了解凭据助手之前，用户可以使用包含桌面环境 keychain 接口的 askpass 程序，例如 kwalletaskpass (KDE Wallet) 和 git-password (MacOS X Keychain)。

## 5.4 发布变更到上游

前面 5.1 “协作工作流”已经介绍过多种版本库的创建方法，我们将会学习一些常见的为项目添加工作成果的方式。接下来将会向读者介绍发布变更的主要方法。

在开始为项目工作之前，用户最好能够经常和主工作流保持同步，将官方版的版本库及时集成到自己的本地版本库中。当然，上述工作是属于维护者的，详情可以参考第 7 章。

### 5.4.1 推送变更到公共版本库

在中心式工作流中，用户只需要将本地的变更推送到中心服务器即可，如图 5-12 所示。因为用户将中心版本库共享给了团队其他成员，也有可能出现用户尝试将变更推送到中心版本库上时，其他成员已经推送了一些变更到分支上（即非快进）的情况。在这种情况下，用户需要先执行 pull 操作（拉取和合并相关变更，或者拉取变更后进行变基操作）集成其他人员的工作成果，然后再推送自己的变更。

另外一个在相同工作流中可能遇到的问题是，你的团队中每个成员都提交了自己的变更集到代码审核系统中，例如 Gerrit。一个可行的解决方案是在特定版本库中推送某个特定的 ref 引用（以目标分支名结尾，例如 refs/for/<branchname>）。然后变更审核服务器可以根据独立的 ref 引用集合自动集成每组变更（例如 refs/changes/<change-id> 是根据一系列的变更 ID 进行提交命名的）。



在对等网络工作流（参见图 5-3）、维护者工作流和层级工作流变体中（参见图 5-4 和图 5-5），第一步在项目获取变更时也是执行推送操作，不过是推送到自己的公共版本库。然后用户需要请求团队其他开发人员或者项目维护者将自己的变更合并。用户可以通过生成一个 pull request 来达到上述目的。

## 5.4.2 生成 pull 请求

在使用个人公共版本库的工作流中，用户需要向合作开发者、维护者或者集成经理发送有新的变更需要推送的通知。`git request-pull` 命令可以帮助用户完成此目的。给定起始点（目标修订区间的起始位置）、远程公共版本库的 URL 或者名称，它将会生成一个变更记录摘要：

```
$ git request-pull origin/master publish
The following changes since commit
ba5807e44d75285244e1d2eacb1c10cbc5cf3935:
```

```
Merge: strtol() + checks (2014-05-31 20:43:42 +0200)
```

```
are available in the git repository at:
```

```
https://git.example.com/random master
```

```
Alice Developer (1):
```

```
Support optional <count> parameter
```

```
src/rand.c | 26 ++++++-----
1 files changed, 21 insertions(+), 5 deletions(-)
```

pull 请求包含基于变更记录的 SHA-1 码（一系列准备执行 pull 操作的首个提交对象之前的修订）、提交的标题、公共版本库的 URL 和分支（作为 `git pull` 命令的相关参数），以及变更的 `shortlog` 和 `diffstat`。

上述输出结果可以发送给维护者，例如通过电子邮件的形式。很多 Git 托管软件和服务都内置了对 `git request-pull` 命令相同的功能（例如 GitHub 上创建 pull 请求的特性）。

## 5.4.3 交换补丁

很多大型项目（以及很多开源项目）都建立了以补丁的形式接受变更记录的流程，例如为了降低贡献变更的准入门槛。如果用户想给某个项目发送一次性的贡献代码，但是又不打算成为正式的项目开发人员，那么发送补丁也许会比创建完整的协作流程更简单一些。

(为了获得向中心工作流提交修订的权限,用户需要为拉取工作流建立个人的公共版本库)。此外,用户可以使用任何兼容的工具生成补丁,项目可以在不考虑用户使用何种版本控制工具类型的前提下接受补丁。



如今,随着各种免费的 Git 托管服务的普及,可能更不容易找到一款可以发送电子邮件格式补丁的邮件客户端了,不过 submitGit (专门提交补丁到 Git 项目的邮件列表) 之类服务应该可以帮上忙。

此外,作为变更记录文本形式的补丁,可以方便地被计算机和人类读取。这使得它们广受用户青睐,而且非常适合进行代码审核。很多开源项目都使用公共的邮件列表来达到此目的:用户可以发送一封电子邮件到该列表,然后公众可以在你提交的变更记录的基础上审核和添加注释。为了将每个提交生成电子邮件的版本转换成 mbox 格式的补丁,用户可以使用 `git format-patch` 命令:

```
$ git format-patch -M -1
0001-Support-optional-count-parameter.patch
```

用户可以将任意修订区间指示符与上述命令一起使用,如前面的示例所述,最常用的方式是限定修订数量,以及使用双点符号表示修订区间,例如 `@{u}..` (详情可以参考第2章)。当生成大批量的补丁时,选择一个目录来保存生成的补丁是非常有用的。用户可以使用 `-o <directory>` 指定目录地址。`-M` 选项在 `git format-patch` 命令(通过 `git diff` 命令)中的作用是启用重命名检测功能,这样做的好处是补丁体积更小,更方便审核。

补丁文件通常是以下列类似内容结尾的:

```
From db23d0eb16f553dd17ed476bec731d65cf37cbdc Mon Sep 17 00:00:00 2001
From: Alice Developer <alice@company.com>
Date: Sat, 31 May 2014 20:25:40 +0200
Subject: [PATCH] Initialize random number generator
```

```
Signed-off-by: Alice Developer <alice@company.com>
```

```
---
```

```
random.c | 2 ++
```

```
1 files changed, 2 insertions(+), 0 deletions(-)
```

```
diff --git a/random.c b/random.c
```

```
index cc09a47..5e095ce 100644
```

```

--- a/random.c
+++ b/random.c
@@ -1,5 +1,6 @@
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int random_int(int max)
@@ -15,6 +16,7 @@ int main(int argc, char *argv[])

    int max = atoi(argv[1]);

+   srand(time(NULL));
    int result = random_int(max);
    printf("%d\n", result);
--
2.5.0

```

这实际上是一封 mbox 格式的完整电子邮件。从主题 (Subject) 开始到---符号之前的内容是注释信息, 即变更记录的说明。为了将它们发送到邮件列表或者一个开发人员那里, 用户可以使用 `git send-email` 或者 `git imap-send` 命令。维护者可以使用 `git am` 命令集成上述一系列补丁, 自动化创建相关的提交对象, 详情可以参考第 7 章。



这里的 [PATCH] 前缀是为了让用户更方便地将它们和其他电子邮件区分开来。前缀可以也经常包含额外信息, 例如一系列补丁的数目、修订的名称、正在研发中的描述信息, 或者请求相关的状态说明, 例如 [RFC/PATCHv4 3/8]。

用户可以帮助将来的审核人员给这些补丁添加更多信息。例如有关替代方法的信息, 补丁和上一修订版本之间的差异 (以前的提交), 或者补丁在代码实现上成员讨论的引用摘要 (例如相关的邮件列表)。用户可以将文本添加到变更摘要之前, 即补丁起始位置和---符号之间。不过上述内容在执行 `git am` 命令时会被忽略。

## 5.5 信任链

项目开发过程中的代码质量是衡量协作效果好坏的重要指标之一, 其中包括防止意外



事件对版本库造成损害、黑客的恶意入侵等，这些问题都是可以借助版本控制系统解决的。Git 需要确保版本中的内容是可信的，其中包括用户自己和其他开发人员（特别是项目的官方版本库）。

### 5.5.1 内容地址存储

第2章已经介绍过 Git 采用 SHA-1 哈希码作为提交对象（代表项目历史的修订记录）的原生标识符。这种机制能够以分布式生成提交标识符，使得提交对象的 SHA-1 加密函数可以指向上一个提交对象（包含父提交的 SHA-1 标识符）。

此外，所有其他存储在版本库中（包括用 blob 对象表示修订对象的文件内容，以及树对象代表的层级文件结构）的数据可以采用上述机制。所有类型的对象都是根据其内容编址的，更精确地说，是对象的哈希函数。用户可以将 Git 版本库的基础结构当作根据内容编址的对象数据库。

Git 通过安全的 SHA-1 码内建了信任链。一方面来看，提交的 SHA-1 码是基于其内容生成的，其中还包括父提交的内容，依次类推直到根提交。另一方面，提交对象的内容包括项目顶层目录树的 SHA-1 码，它根据相关内容生成，这些内容又包含子目录树和文件内容的 blob 对象，依次延伸到每个独立文件。

上述特性使得 SHA-1 码可以用来验证对象来源（存在不信任因素）的合法性，即它们自创建以来是否被非法篡改过。

### 5.5.2 轻量级标签、附注标签和签名标签

信任链可以帮助用户验证内容，不过无法验证创建该内容的用户的标识符（作者和提交者的名字都是可以进行配置的）。GPG/PGP 签名就是为了解决这一问题的，其包括签名标签、签名提交和签名合并。

Git 支持两种标签：轻量级标签和附注标签。

#### 轻量级标签

轻量级标签和分支很像，并不修改变更记录，它只是修订图上指向特定提交节点的指针（引用），因此在 refs/tags/命名空间中的数据比在 refs/heads/中功能更强。

#### 附注标签

附注标签，顾名思义，也包含标签对象。这里的标签引用（在 refs/tags/种）指向

了标签对象，然后该对象指向某个提交对象。标签对象内包含其创建日期、标签化的标识符（名称和电子邮件），以及一个标签化的信息。用户可以使用 `git tag -a`（或者 `-annotate`）命令创建一个附注标签。如果用户没有在命令行中为附注标签声明信息（例如使用 `-m "<message>"` 选项），Git 将会自动打开用户的编辑器，以方便用户输入信息。

用户可以使用 `git show` 命令查看标签化提交中的标签数据（注释已略去）：

```
$ git show v0.2
tag v0.2
Tagger: Joe R Hacker <joe@company.com>
Date: Sun Jun 1 03:10:07 2014 -0700

random v0.2

commit 5d2584867fe4e94ab7d211a206bc0bc3804d37a9
```

## 签名标签

签名标签是附有明文 GnuPG 签名的附注标签。用户可以使用 `git tag -a` 或者 `git tag -u <key-id>` 命令创建它（使用开发者自己的提交者标识符选择签名密钥，或者使用 `user.signingKey` 设置），上述两种方式都假定用户拥有私人的 GPG 密钥（创建时可以使用 `gpg--gen-key` 选项）。



附注或者签名标签的诞生通常意味着软件有新版本发布，轻量级标签代表私人或者临时的修订标签。为此，某些 Git 命令（例如 `git describe`）将默认忽略轻量级标签。

当然，在协作工作流中将签名标签公开是非常重要的，而且这也是一种验证方式。

## 发布标签

默认情况下，Git 系统是不会推送标签的，用户必须明确声明。一种解决方案是使用 `git push <remote> tag <tag-name>` 命令独立推送标签（标签符号 `<tag>` 和更长的 `refspec` 的符号 `refs/tags/<tag>:refs/tags/<tag>` 是等效的）；当然，大部分情况下，用户还可以忽略该标签符号。另外一种解决方案是使用 `--tags` 选项大批量推送轻量级标签和附注标签，或者使用 `--follow-tags` 选项只推送指向提交对象的附注标签。

很明显，当发现提交对象被标记了错误的标签时，又或者标签是非公开的时，用户可以随意地对标签进行编辑（使用 `git tag -f` 命令）。

当用户更新变更时，Git 会自动遵循标签的规则，下载附注标签指向的提交节点对象。这意味着下游开发者将会自动获取签名标签，并且能够对预览版本进行校验。

## 标签验证

为了对签名标签进行验证，用户可以使用 `git tag -v <tag-name>` 命令。为此用户还需要将签名者的 GPG 公钥添加到自己的密钥环中（可以通过 `gpg --import` 或者 `gpg --keyserver <key-server> --recv-key <key-id>` 等命令导入），当然标签生成者的密钥也能够通过用户自身信任链的审查。

```
$ git tag -v v0.2
object 1085f3360e148e4b290ea1477143e25cae995fdd
type commit
tag signed
tagger Joe Random <jrandom@example.com> 1411122206 +0200
```

```
project v0.2
gpg: Signature made Fri Jul 19 12:23:33 2014 CEST using RSA key ID
A0218851
gpg: Good signature from "Joe Random <jrandom@example.com>"
```

## 5.5.3 签名提交

签名标签是用户和开发人员校验发布的相关标签是否是维护者创建的一个很好的解决方案。不过如何确认一个提交是某个名为 Jane Doe、电子邮件是 `jane@company.com` 的人创建的呢？如何让任何人都可以对上述情况进行校验检查呢？

一个可选的解决方案是 Git 自 1.7.9 版本以来支持的 GPG 签名的独立提交。用户可以使用 `git commit --gpg-sign[=<keyid>]` 命令（或者 `-S` 的缩写形式）达到上述目的。密钥标识符是可选的，如果没有提供该标识符，Git 将会自动采用用户的标识符。注意，`-S` 选项和 `-s` 选项是有区别的，后者会在提交信息尾部添加一行经过数字证书所有权审核的验证结果信息。

```
$ git commit -a --gpg-sign
```



```
You need a passphrase to unlock the secret key for
user: "Jane Doe <jane@company.com>"
2048-bit RSA key, ID A0218851, created 2014-03-19
```

```
[master 1085f33] README: eol at eof
1 file changed, 1 insertion(+), 1 deletion(-)
```

为了让提交支持校验，只需推送它们即可。任何人希望对它们进行校验的话，都可以通过 `git log`（或者 `git show`）命令配合 `--show-signature` 选项；在 `git log --format=<format>` 中添加 `%Gx` 占位符也能达到同样的目的。

```
$ git log -1 --show-signature
commit 1085f3360e148e4b290ea1477143e25cae995fdd
gpg: Signature made Wed Mar 19 11:53:49 2014 CEST using RSA key ID
A0218851
gpg: Good signature from "Jane Doe <jane@company.com>"
Author: Jane Doe <jane@company.com>
Date:   Wed Mar 19 11:53:48 2014 +0200
```

```
README: eol at eof
```

自 Git 2.1.0 版本以来，用户还可以使用 `git verify-commit` 命令对相关提交进行校验。

## 5.5.4 合并签名标签（合并标签）

前文所述的签名提交机制在某些工作流中可能会非常有用，不过在用户的早期推送过程中可能会带来不便，例如在用户自己的公共版本库中，经过一段时间后，用户会对是否值得将它们推送到上游（是否知道发送到主版本上）而举棋不定。如果用户遵循第 8 章推荐的方法，上述情况是可能发生的，而且只会在事后（提交创建很久之后）被察觉，即系列迭代提交经过几轮代码审核之后。

用户可以通过它的图形闭合之后（通过代码审核之后）重写整个提交集合来解决这个问题，对每个重写后的提交进行签名，或者只对顶层的提交进行修改和签名。上述两种方法都需要强制推送提交以替换签名的提交历史记录。用户还可以创建一个空的提交对象（使用 `-allow-empty` 选项），对它签名，然后将它推送到顶层提交记录上。不过还有一个更好的解决方案——请求拉取签名标签（Git 1.7.9 以及以上版本支持）。



在这种工作流中，用户可以进行功能特性的开发，当它们达到预期目标之后，就可以创建一个签名标签并将它们推送到服务器上（在一系列开发工作中最后一个提交上添加标签）。用户不需要将自己的工作分支推送到服务端——只需推送标签即可。如果工作流中出现了一个发送给集成管理者的拉取请求，用户可以使用一个标签作为其最终的提交节点：

```
$ git tag -s for-maintainer
$ git request-pull origin/master public-repo 1253-for-maintainer \
>msg.txt
```

签名标签的信息会在 pull 请求中用虚线与其他内容区分开，这意味着用户在创建签名标签时可以在标签信息中添加和该工作有关的解释性信息。维护者在收到这样的 pull 请求之后，可以根据它拷贝相关的版本库，获取和集成相关的具名标签。当记录汇总拉取请求的具名标签之后，Git 将会打开用户的编辑器，并要求用户输入注释信息。集成人员将会看到以如下内容开始的模版：

```
Merge tag '1252-for-maintainer'

Work on task tsk-1252

# gpg: Signature made Wed Mar 19 12:23:33 2014 CEST using RSA key ID
A0218851
# gpg: Good signature from "Jane Doe <jane@company.com>"
```

这个提交模版包括被合并签名标签对象的 GPG 验证输出结果注释（不过这并不是最终的合并操作的注释信息）。标签信息有助于用户更好地描述合并操作。

签名标签并不是和标签对象那样被拉取后存放在集成者的版本库中的，它的内容其实是被隐藏存放在合并提交对象里的。这么做的原因是不希望这类工作标签因为数量庞大的原因而污染标签的命名空间。在签名标签被集成之后，开发者可以安全地将这些标签删除（例如 `git push public-repo --delete 1252-for-maintainer`）。

记录合并提交内部的签名可以方便地使用 `--show-signature` 选项进行事后验证：

```
$ git log -1 --show-signature
commit 0507c804e0e297cd163481d4cb20f3f48ceb87cb
merged tag '1252-for-maintainer'
gpg: Signature made Wed Mar 19 12:23:33 2014 CEST using RSA key ID
A0218851
gpg: Good signature from "Jane Doe <jane@company.com>"
Merge: 5d25848 1085f33
```

Author: Jane Doe <jane@company.com>

Date: Wed Mar 19 12:25:08 2014 +0200

Merge tag 'for-maintainer'

Work on task tsk-1252

## 5.6 小结

我们已经学习了如何使用 Git 软件进行团队协作开发，如何在一个团队中一起推进一个项目。我们介绍了多种协作工作流以及多种方便协作的建立版本库的方式。需要因地制宜地决定使用哪种协作工作流，例如考虑团队规模大小、类别如何等。本章主要精力聚焦于版本库和版本库之间的交互。版本库中本地分支和远程跟踪分支之间的交互将会放在第 6 章具体讨论。

我们还学习了在选定协作工作流的情况下，如何使用 Git 管理远程版本库的信息，还了解了如何存储、查看和更新这些信息。本章还介绍了管理不规则工作流的方法，用户可以从某个版本库（官方版本库）拉取信息，同时可以将变更记录推送到其他版本库（公共版本库）。

我们还学习了在远程服务器支持的情况下如何选择传输协议，以及使用原生 Git 版本库时访问外源版本库的技巧。

访问远程版本库时需要提供凭据，一般来说这些内容是用户名和密码，例如推送变更记录到远程版本库。本章还介绍了在 Git 中使用凭据助手如何更方便地帮助用户完成上述步骤。

发布你的变更并推送到上游，其中根据工作流的不同可能还会使用若干不同机制。本章主要介绍的是推送、pull 请求和基于补丁的技术。

我们学习了信任链的概念：如何对集成者发布的软件版本进行校验，如何给自己的工作成果签名以方便集成者校验其合法性，以及 Git 系统架构对上述机制的辅助作用。

接下来的两章将会对团队协作这个主题进行进一步扩展：第 6 章将会向读者介绍本地分支和远程版本库上的分支之间的关系，以及如何为团队协作建立分支，第 7 章将会讲述与此相反的主题——如何进行并行开发。

## 第 6 章

# 分支应用进阶

上一章主要讲述的是如何进行团队协作，重点放在了版本库层面的交互。该章主要向读者介绍了多种中心式和分布式工作流，以及它们的优缺点。

本章将会深入介绍分布式开发的团队协作，其中会对本地分支和远程版本库上的分支之间的关系做概要介绍；同时引入了远程跟踪分支、分支跟踪和上游等概念。本章还会向读者介绍如何使用 `refspecs` 和推送模式在不同版本库之间进行分支同步。读者还会学到一些分支技术，如何在发布软件预览版和修复 `bug` 时使用分支。同时本章还会介绍如何使用分支的特性更方便地为软件的下一版本增减功能特性。

本章的主要内容包括以下几个方面。

- 长期分支和短期分支的类型以及用途。
- 多种分支模型，其中包括基于工作流的主题分支。
- 不同分支模型的发布流程。
- 在多个预览版程序中使用分支修复安全问题。
- 远程跟踪分支和 `refspecs` 规范，以及默认远程版本库配置。
- 拉取、推送分支和标签的规则。
- 为协作工作流选择适当的推送模式。

### 6.1 分支的类型和用途

分支在版本控制系统中代表一系列的并行开发工作。接下来我们将会看到，它在隔绝

和分离不同工作方面是非常有用的。例如分支可以用来阻止用户当前开发某个特性的工作对 bug 修复方面的影响。

单个 Git 版本库可以拥有任意数量的分支。不过在类似 Git 这样的分布式版本控制系统中，可能会在单个项目下存在多个版本库（forks），其中有些是公开的，有些是私有的，每个版本库都会有自己的本地分支。在分支层面了解版本库之间的团队协作之前，我们需要知道可能遇到的本地和远程版本库的分支类型。接下来的内容将会介绍如何使用这些分支，以及人们在单个版本库中使用多个分支的原因。

### 历史回顾：分支管理的演变

早期的分布式版本控制系统在每个版本库模型中只允许使用一个分支。Bazaar（后来的 Bazaar-NG）和 Mercurial 官方文档在其诞生初期就向用户建议在克隆版本库时新建一个分支。

另外一方面，Git 软件从发布之初就支持在单个版本库中使用多个分支。不过在刚发布时，它的架构是一个中心式多分支版本库和其他单分支版本库进行交互（例如，帮助手册中 gitrepository-layout(7)介绍的普通目录.git/branches 指定 URL 和拉取分支），不过 Git 的出现是对以往传统软件的颠覆。

因为在 Git 中创建分支非常简单（合并操作也很容易），使用它协作开发也非常灵活，因此其逐渐受到开发人员的青睐，即使是独立开发工作也是如此。这导致了非常实用的主题分支工作流的普及。



将一系列的某类开发工作隔离的原因很多，这也是导致了多种分支的出现。不同类型的分支的用途也各不相同。某些分支是长期的，甚至是永久的，同时某些分支是短期的并且它们在失效之后还可能被删除。某些分支专门用于发布软件，某些分支则不是。

## 6.1.1 长期或者永久性分支

长期或者永久性分支是为了保证持续性而引入的（永久性从某种意义上来说至少是很长的一段时间）。

从协作开发的角度来看，长期分支可以作为用户下一次更新数据或者发布变更的基础。



这意味着用户可以放心地从远程版本库长期分支的任意位置（fork）开始他们自己的工作，并且可以确保将自己的工作成果方便地整合。

当然，用户在公共版本库中可以找到的只有长期分支。大部分情况下，这些分支将永远不会被重置（新版本总是由旧的版本衍生而来）。不过也有一些特殊情况，新的预览版程序发布之后，某些分支会被重新构造（此时需要执行强制拉取操作），某些分支可能是快进式的。上述特例都会在开发者文档中明确提及，防止用户碰到不必要的麻烦。

## 集成、节点或者渐进稳定性分支

将正在开发的一系列工作（其中可能包括临时的不稳定代码）和维护工作（其中的内容只包括 bug 修复）隔离是分支的用途之一。通常这类分支或多或少都存在一些。这种分支的主要目的是将和稳定性有关的开发工作整合起来，其涵盖范围包括项目维护、稳定版程序、测试版程序、项目开发等多个环节。

这些分支来自稳定性依次递减的层级工作成果集合，如图 6-1 所示。需要注意的是，在实际开发工作中，渐进稳定性分支的存在形式不会像图片展示的那样简单。在拉取节点之后的分支上也可能出现新的修订记录。不过即使发生过合并操作，大体的框架仍是一致的。

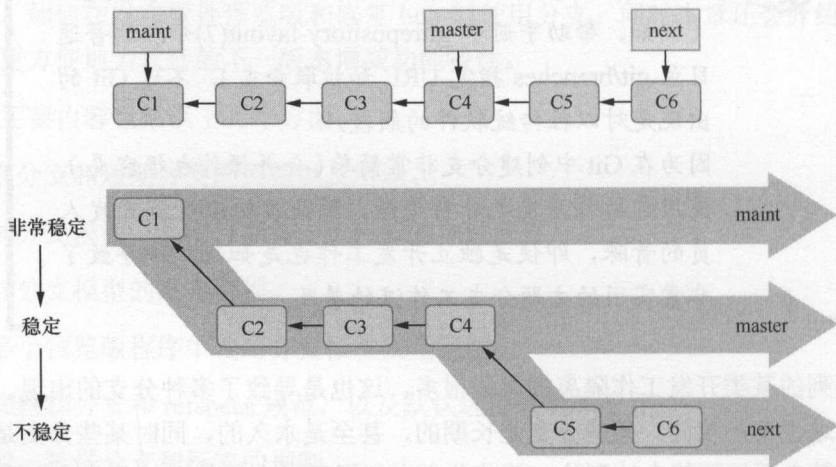


图 6-1 线性视图和“筒仓”视角下的渐进稳定性分支。在线性视图中，越稳定的修订版本在提交历史中记录历史越久远，不太稳定的修订记录在提交历史中离当前时间更近。或者，我们可以将分支当作流水线研发工作的进展依赖变更记录的稳定程度（节点）

一般的规则通常是将更稳定的分支合并到较不稳定的分支中，即向上合并，这使得分支可以保持整体的筒仓造型（参考图 6-2 或者 6.2.2 “节点或者渐进稳定性分支工作流”）。这是因为合并操作通常意味着将被合并分支的所有变更都添加到目标分支中。因此将一个

不太稳定的分支合并到一个更稳定的分支中会对稳定的分支产生不良影响，有悖于稳定分支建立的初衷。

一般来说，我们可以从节点分支上看到以下不同层次的稳定性：

- 用于修复 bug 的维护性分支，其中只包含和最新的主版本软件有关的 bug 修复；次版本发布的软件可以基于该分支进行构建。
- **master** 或者稳定主干分支，包含将要发布的下一主版本的所有开发工作；该分支的外部引用应该一直处于准发布状态。
- **next** 或者不稳定分支，对新开发的功能进行测试，方便用户验证是否能够将该特性集成到下一版本中。该分支的外部引用可以用于每日构建。
- 特性测试分支，专门对新特性进行集成测试的分支，主要测试软件不同特性之间的兼容性。

保留多个长期分支并不是必需的，不过对于大型或者复杂的项目来说经常会很有帮助。通常在操作层面，每个级别的稳定性是和其自身平台、开发环境和给定分支的平台一一对应的。

### 每一版本分支和每一版本的维护

为发布项目新版本而做的准备工作是漫长而复杂的。每一版本分支可以为用户提供不少帮助。这类版本分支可以将正在进行的研发工作与将要发布的新版本程序隔离。这样一来，其他开发人员可以继续对新特性的研发和集成测试，同时质监部门在版本管理人员的帮助下可以对候选版本进行稳定性测试。

在创建一个新的软件版本之后，保留这类每一版本分支可以让我们方便地维护软件以往的候选版本。在此期间，这类分支可以用来当作存放修复 bug 提交的地方，以及创建候选版本软件的镜像。

并不是每个项目都可以找到每一版本分支的用武之地。用户可以在稳定节点分支上为一个新的候选版本做准备，或者使用一个独立的版本库代替使用独立分支。当然，并不是所有项目都需要提供除最新版本之外的多版本支持。

这类分支的命名通常是和新发布的版本有关的，例如 `release-v1.4` 或者 `v1.4.x`（命名时最好不要和候选版名称对应的标签重名）。

### 处理安全漏洞的 Hotfix 分支

Hotfix 分支和候选版分支类似，不过其是计划外的候选版本。它们的用途是处理某个

上线产品或者广泛部署的版本出现意外的情况，通常是为了解决上线产品中某些关键的问题（一般是一个比较严重的安全漏洞）。这类分支可以被看作一个长期的 bug 修复主题分支（详情可以参考本章 6.1.2 节的“bug 修复分支”部分）。

### 每一用户或者每一开发分支

如前所述，开发者的项目客户因为有一些特别的需求，需要项目支持某些定制特性；又或者用户在部署网站时需要一些特殊的条件。假如这些需求无法通过简单地修改配置文件来满足，那么开发者最好为这些客户定制需求创建一个独立开发分支。

但是开发者又不希望将上述开发工作和其他工作一直保持隔离，而是希望在适当的时机将它们和主分支合并。一种方案是为每一定制特性集合、每一客户或者每一开发创建一个分支；另外一种方案是使用独立的版本库。上述两种方案都支持并行开发，并且迁移变更也很方便。

### 自动化分支

假定读者正在研发一个 Web 项目，希望使用版本控制系统实现网站部署的自动化。一种解决方案是建立一个监控程序监控特定分支的变化（例如名为“deploy”的分支）。该分支被更新的同时自动更新和重新加载网站应用。

当然，不只有这一种解决方案。另外一种解决方案是使用独立的部署版本库，并建立对应的钩子，当执行推送操作时同时触发更新网站应用的操作。或者用户可以在公共版本库上配置一个钩子，以便在推送某个特定分支时触发重新部署网站应用的指令（该机制的详情可以参考第 11 章）。

这类技术除了可以用于部署应用之外，还可以用于持续集成（Continuous Integration, CI），推送更新到某个特定分支后会触发运行测试用例的操作（触发器可以通过在该分支上创建一个新提交或者从其他分支合并而来）。

### 支持匿名推送的群体性分支

远程版本库中有一种分支可以对推送操作进行特殊处理，用到这种技术的很多地方很多，其中就包括辅助协作开发。它可以用来为某个项目启用匿名推送访问控制。

假定你允许访客推送变更到中心版本库，又或者希望使用托管的方式达到此目的，一种解决方案是创建一个支持类似访问控制的特定群体性（mob）分支或者一个包含 mob/\* 的命名空间（一组分支）。



详情可以参考第 11 章的内容。

## 孤儿分支

目前为止，向读者介绍的所有分支在用途和管理方式上都各不相同。从技术层面上看（从提交视图上），它们都是一样的。不过孤儿分支是个特例。

孤儿分支是并行无关联的一系列开发工作，即它们和项目主线历史修订没有共享的修订。在 DAG 视图中它们是用无关联的子图来表示的，并且和主 DAG 图形没有任何交互关系。大部分情况下，它们签出的文件也是不相关的。

这类分支也是一种在单个版本库中存放不相关内容的技巧，它可以看作版本库隔离的替代性方案（在使用独立的版本库存放各自独立的内容时，用户也许会使用某些命名约定来修饰版本库名称，例如相同的前缀）。

它们常见的用法有以下几种：

- 存放项目的 Web 页面文件，例如 GitHub 使用名为 `gh-pages` 的分支存放项目的 Web 页面。
- 在编译代码时需要用到一些非标准的工具链时存放中间文件。例如项目帮助文档可以分别存放在 `html`、`man` 和 `pdf` 等孤儿分支中（`html` 分支也可以用来发布文档）。通过这种方法，客户无须安装相关的工具链就能获取它们了。
- 存放项目计划（TODO list）信息（例如在 `todo` 分支中），也许还会存储一些特殊的维护工具（脚本）。

用户可以使用 `git checkout --orphan <new branch>` 命令创建这类分支，也可以通过独立的版本库向特定的分支推送（或者拉取）：

```
$ git fetch repo-htmldocs master:html
```



使用 `git checkout --orphan` 命令创建一个孤儿分支时，从技术上来说并没有新建一个分支引用。实际的行为是将 HEAD 标识符引用指向了一个未出生的分支，即在新的孤儿分支上添加第一个提交对象之后创建了上述引用。

这也是 `git branch` 命令创建一个孤儿分支时没有可用的选项参数的原因。



## 6.1.2 短期分支

在长期分支是为了持续性而存在的同时，创建短期或者临时分支的目的通常是为了解决单个问题，并且相关问题解决之后，它们通常会被删除。它们存在周期的长短是和解决相关问题所花费的时间密切相关的。它们的目标是有时间限制的。

因为它们的临时性，它们常见于开发者或者集成管理者（维护人员）的本地私人版本库，并且不会被推送到分布式的公共版本上。如果它们出现在了公共版本库上，那么它们只可能出现在独立开发者的公共版本库中，被当作某个 pull 请求的目标。

### 主题或者特性分支

分支常用来隔离和集成开发工作的不同子集。因为分支和合并都非常简单，我们甚至可以像前文所述那样，进一步根据稳定性级别创建对应的分支。我们还可以为每个独立的问题创建独立分支。

主题分支目标是为每个主题创建一个新分支，即一个新特性或者一个 bug 修复。这种类型的分支的目的既包括后续开发工作功能特性的整合（每一个提交应该是自包含结构，方便日后代码审核），也要包括将开发某个特性的工作和其他主题隔离开来。采用主题分支后，特性相关的变更可以保持一定的独立性，避免和其他提交混杂在一起；同时也方便用户将整个分支当作一个单元统一移除（或者回退）、审核和集成变更。

主题分支上的提交记录最终目标是涵盖产品的某个预览版本。这意味着短期分支最终会被整合到长期分支中，即渐进稳定性工作成果收集，之后被删除。为了方便地集成主题分支，推荐的做法是通过最原始、最稳定的版本创建这类分支，并最终将该主题分支与之合并。通常这意味着从一个渐进稳定性分支创建一个新的主题分支。不过如果给定特性依赖的主题不在稳定分支中的话，用户还需要添加相应的依赖引用到主题分支中。

注意，如果事后证明用户提取了错误的分支，还可以通过变基操作解决（详情可以参考第7章和第8章），因为主题分支并不是公开的。

### bug 修复分支

读者很容易区分出用于 bug 修复的这类特殊的主题分支。这种分支应该是从它适用的最原始的集成分支(包含 bug 的最稳定分支)创建的。这通常代表用户提取的内容是维护分支或者剔除了所有集成分支，这一点是和稳定分支的外部引用不同的。bug 修复分支的目标是被整合到相对长期性的集成分支中。

bug 修复分支可以被当作一个短期的长期 hotfix 分支等价物。

使用它们的效果远好于简单地在维护分支（或者相应的集成分支）上提交修复提交。

## 脱离 HEAD——匿名分支

读者可以将 HEAD 分离的状态当作临时分支的极端情况，以至于这种分支甚至都没有名字。在少数情况下，Git 会自动使用这种匿名分支，例如在执行二分查找和变基操作时。

因为在 Git 中只允许出现一条匿名分支，并且该分支还必须是用户当前使用的分支。因此，比较推荐的做法是创建一个包含临时名称的临时分支，用户在后续开发过程中还可以修改上述临时分支的名称。

脱离 HEAD 的应用场景之一是某些概念的验证性工作。当然，如果事后证明这些验证性工作是有价值的，用户还需要给这个分支设置适当的名称（或者用户还需要进行分支切换）。从一个匿名分支向一个具名分支的切换非常简单。用户只需要根据当前脱离 HEAD 的状态创建一个新分支即可。

## 6.2 分支工作流和发布工程

现在我们已经了解了分支的种类和用途，接下来介绍一些分支的用法。需要注意的是，用户应该因地制宜地使用不同分支。例如，小型项目最好使用更简单的分支工作流，而较大的项目则可能需要使用更高级的工作流。

本小节将会向读者介绍多种标准工作流的使用。每种工作流可以通过采用的不同分支类型（关于分支类型前面章节已经介绍过）进行识别。我们除了可以了解给定分支中正在进行开发的工作组织形式之外，还可以知道在将要发布软件新版本时需要完成哪些工作（分清工作内容的主次）。另外，本小节还会介绍选定工作流之后分支内部的演化过程。

### 6.2.1 预览或者主干分支工作流

最简单的工作流之一是只使用一个独立的集成分支。这类分支有时也叫主干，在 Git 系统中，它们通常就是 **master** 分支（创建版本库时的默认分支）。在这种工作流以纯粹的版本记录来看，至少在平常的开发工作中，用户应该将所有提交添加到上述分支上。这种工作方式是由使用中心式版本控制决定的，它适合处理分支和合并操作花费代价太大，但是用户希望避免处理过多分支的情况。



这种工作流更高级的形式中，用户还可以使用主题分支和每个特性的短期分支，并在后续工作中将上述分支合并到主干分支中，从而替代直接在主干分支添加提交的做法（参见图 6-3）。

在这种工作流中，当软件新的发行版本确定后，我们可以在主干分支之外创建一个新的预览版分支。这么做的目的是避免给预览版稳定性和正在开发的工作造成不良影响。原则是所有和稳定性相关的工作继续在预览版分支上推进，同时所有研发工作放在主干分支上进行，将要发布发行预览版程序从预览版分支上提取（添加标签），并将之作为新版本程序的最终版本。

对于给定版本的预览版分支，用户后续还可以用它来收集 bug 修复方案和提取次要版本。

这类简单工作的不足之处在于，在开发过程中，用户经常会碰到软件不稳定的状态。这种情况下，选取一个足够稳定的节点发布新版本将会变得很困难。一种替代性的解决方案是在预览版分支上创建一个恢复性提交，将工作内容恢复到发布前的某个状态。但是这样一来工作量会非常大，并且项目历史记录也会变得难以回溯。

这种工作流的另外一个不足之处就是，某些特性乍一看会让人眼前一亮，但是后续使用过程中会带来不少麻烦。这也是采用这种工作流后需要处理的问题。如果在开发过程的后期发现为某些特性创建多个特性提交记录并不是一个好主意，恢复它们到以前的状态将会非常困难，尤其是当它们的提交遍布项目历史的时间线时。

此外，主干和预览版分支工作流也没有为查找不同特性之间的不良交互提供固有机制，即集成测试。

尽管存在这些问题，这种简单的工作流对于小型团队来说仍是非常合适的。

## 6.2.2 节点或者渐进稳定性分支工作流

为了能够提供一系列性能稳定的软件产品和在实践中对其进行测试，并将其当作一种浮动性的测试版本，用户需要将程序稳定性的工作与处于测试阶段的研发工作和不稳定代码隔离。这就是节点分支的用途所在——整合软件不同程度的稳定性和成熟的部分（这种分支有时也称集成分支或者渐进稳定性分支）。如图 6-1 所示，通过本章前面讲述的 6.1.1 节的“集成、节点或者渐进稳定性分支”部分，我们可以看到渐进稳定性分支和线性历史



简易演示代码的图形和筒仓视图。接下来将要介绍采用这种分支的节点分支工作流。

除了将稳定的和不稳定的开发工作保持隔离之外，有时还需要一个持续进行的维护性分支。如果产品只有一个版本需要维护，并且创建新的预览版过程足够简单，用户还可以像上文所述那样采用节点型分支。



这里说的足够简单的意思是指用户可以在稳定分支之外只创建下一个主预览版分支。

在这种情况下，用户至少会拥有 3 个集成分支。一个分支可能会是处理日常维护工作（只包含最新版本的 bug 修复记录），用于创建次要版本。另一个分支是处理稳定性工作，用于创建主版本；该分支还可以用于每日稳定版程序的构建。最后一个分支是用于日常开发工作，可能不太稳定。

在采用这种工作流时可以只使用集成分支。如果有必要的话，可以在维护分支上提交 bug 修复提交记录，然后将它们合并到相应的稳定性分支和开发分支中。用户可以根据通过完备测试的工作成果在稳定性分支上创建提交，然后根据需要再将它们合并到开发分支上（例如新的开发工作依赖上述开发工作的话）。可以将处于测试阶段的开发工作放在开发分支上。在日常开发工作中，用户最好永远不要将不稳定的工作成果合并到更稳定的分支上，否则可能会影响相关分支的稳定性。比较推荐的做法永远是将较稳定的分支合并到较不稳定的那一个上，如图 6-2 所示。

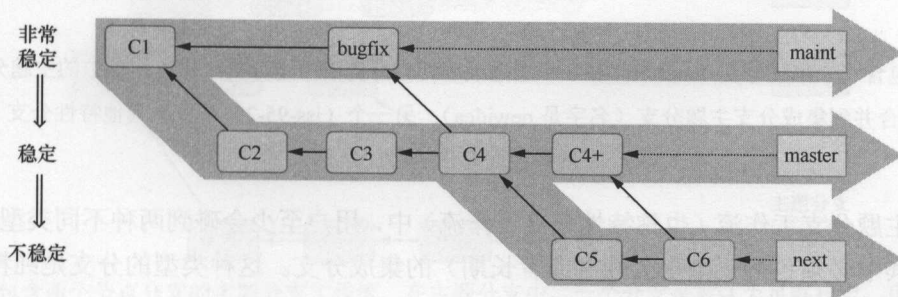


图 6-2 节点或渐进稳定性分支工作流。用户最好永远不要将不太稳定的分支合并到稳定分支中，因为合并操作会影响整个历史记录

当然，这也需要用户事先拥有对正在研发的功能特性是否稳定的研判能力。当然还存在这样的潜在假设，那就是项目之初各功能特性之间能够完全兼容。用户对实际的预期是美好的，不过每个开发特性从理想的萌芽到结出现实之果的过程，在上述特性臻于完美之



前，都需要经历严格试验和激烈的讨论。这个问题可以通过引入下文将要介绍的主题分支来解决。

在纯节点分支工作流中，用户可以在维护分支之外创建次版本分支（包含 bug 修复提交），可以在稳定版本分支之外创建主版本分支（包含新功能特性）。在一个主版本发布之后，稳定性工作分支被合并到维护分支后就可以为刚创建的新版本分支提供支持了。从这一点来看，一个不稳定（开发）分支可以被合并到一个稳定分支中。这也是在上游合并操作中唯一出现的不稳定版本被合并到更稳定版本的情况。

### 6.2.3 主题分支工作流

如图 6-3 所示，主题分支工作流背后的理念是为每个主题创建一个独立的短期分支，以便所有提交都从属于特定分支（相关的开发工作都通过分支进行分类）。每个分支的目的明确，即开发某个新特性或者创建一个 bug 修复提交。

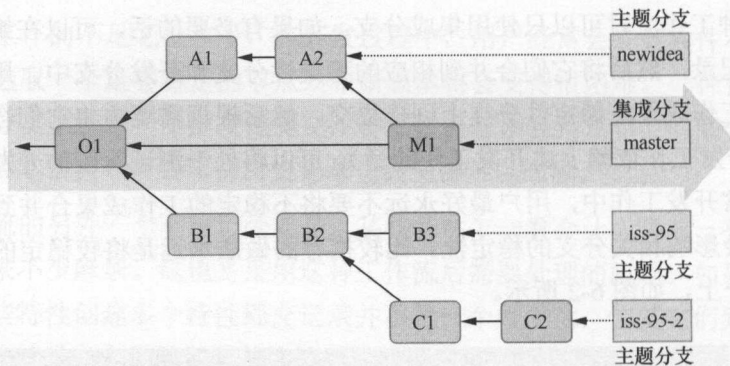


图 6-3 包含一个集成分支（master）和 3 个主题或者特性分支的主题分支工作流。其中的主题分支包括：一个已经合并到集成分支主题分支（名字是 newidea），另一个（iss-95-2）是依赖其他特性分支（iss-95）的特性而存在的特性分支

在主题分支工作流（也称特性分支工作流）中，用户至少会碰到两种不同类型的分支。首先，其中必须包含一个永久性（或者长期）的集成分支。这种类型的分支是纯粹用来进行合并操作的。集成分支是公开的。

其次，其中还包含相互独立的临时特性分支，每个临时分支的目的是开发某个特性或者对某个 bug 进行修复。它们用于执行包含某个特性或者 bug 修复相关的所有步骤，或者是一个开发人员负责的某类工作。当上述特性或者 bug 修复工作完成并被合并到集成分支后，这些临时分支就可以被删除了。主题分支通常是不公开的，也不会出现在公共版本库上。

当某个特性准备进行审核时，它的主题分支通常会执行变基操作方便集成，同时也使得提交历史记录结构更清晰，然后它们会被当作一个整体一起审核。主题分支可以被添加到 pull 请求中，或者作为一系列的补丁被发送（例如使用 `git format-patch` 和 `git send-email` 命令）。为了方便查看和管理，它们通常会被当作独立的主题分支存放在维护者的工作版本库中（例如它们被当作补丁发送时，可以使用 `git am --3way` 命令）。

然后，集成经理（“推举”版本库工作流中的维护人员，或者中心式版本库工作流中的其他开发人员）会审核每个主题分支并且决定这些分支是否符合集成到特定集成分支的条件。如果符合条件，那么它们将会被合并（也许还需要使用 `--no-off` 选项）。

### 主题分支工作流中的节点分支

分支工作流最简单的变体是只使用集成分支，不过通常用户会采用包含主题分支的节点分支工作流这种组合。

在这种常用的变体中，除非分支需要依赖其他特性，特性分支一般是根据给定的稳定分支的外部引用或者最新的主版本程序构建的。在后一种情况下，分支需要从与之相关的依赖项的基础进行构建，例如图 6-4 中的 `feat` 分支。`bug` 修复分支是根据维护性分支构建的。

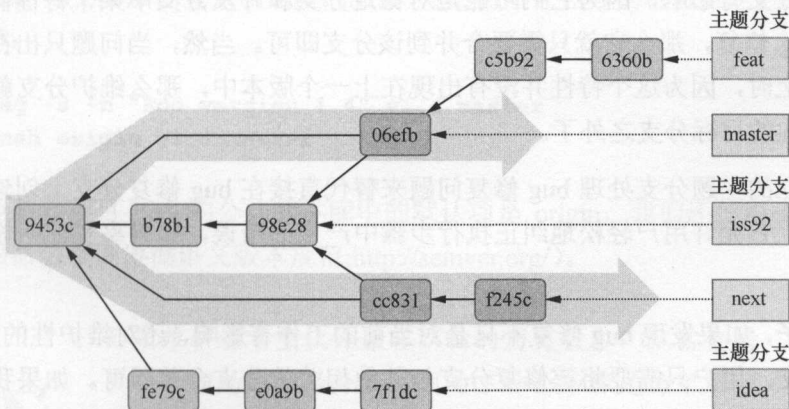


图 6-4 包含两个节点分支的主题分支工作流。在主题分支中，一个分支是被认为足够稳定，可以被合并到 next（不稳定）和 master（稳定）两个节点分支上；另一个分支（idea）已经合并到 next 分支，主要用于测试，以及刚从 master 分支创建的新分支（feat）

当主题完成后，它们首先会被合并到开发集成分支（如 next 分支）上进行测试。如图 6-4 所示，主题分支 idea 和 iss92 都被合并到 next 分支了，但同时 feat 分支还处于研发状态。比较激进的用户可以使用给定的不稳定分支抢先尝试新功能了，当然，同时他们

还需要承担程序崩溃和数据丢失的风险。

经过检验后，当上述特性被认为可以添加到下一个软件版本时，它们将会被合并到稳定性工作的集成分支（例如 `master`）上。图 6-4 就包含这一个分支——`iss92`。在这一点上，在将它们合并到稳定性集成分支上之后，该主题分支就可以被删除了。

使用一个特性分支可以让和该主题相关的修订版本聚集到一处，不用和其他提交记录混淆。主题分支工作流还可以方便地将主题作为一个整体还原，以及同时移除一系列有问题的提交记录（将有问题的提交记录作为一个整体一并移除），用户就不必进行逐个还原操作了。

如果该特性分支事后证明并不成熟，因为它并没有简单地被合并到稳定分支上，只存在于开发分支上。当然，如果我们发现问题为时已晚的话，特性分支已经合并到稳定分支上了，那么还可以回退该合并操作。这是一个比回退单个提交更高级一点的操作，不过它比逐个回退单个提交给人带来的痛苦要少很多，因为用户需要确保每个被回退的提交都执行了正确的操作。回退合并操作的详细操作可以参考第 8 章。

主题分支工作流中包含 `bug` 修复分支也是很常见的。唯一的差异是用户需要考虑将 `bug` 修复分支合并到哪一个集成分支中。当然，这种事情需要用户随机应变。也许 `bug` 修复分支只对维护分支有影响，因为它们可能是对稳定分支和开发分支中某个特性偶尔出现的问题相关的 `bug` 修复，那么它就只需要合并到该分支即可。当然，当问题只出在稳定性分支和开发分支上时，因为这个特性并没有出现在上一个版本中，那么维护分支就被排除在需要合并该特性的目标分支之外了。

使用独立的主题分支处理 `bug` 修复问题来替代直接在 `bug` 修复分支上创建提交还有另外一个好处。它允许用户轻松地纠正执行步骤中产生的错误，如果事实证明修复的分支超出预期的话。

举个例子，如果发现 `bug` 修复不只是对当前的工作有影响，还对维护性的版本有影响，通过主题分支，用户只需要将该修复分支与其他相关的分支合并即可。如果我们直接在稳定性分支上提交这个修复性提交的话则不属于这种情况。对于后一种情况，用户是无法使用合并的，因为这会影响维护分支的稳定性。用户需要拷贝包含修复记录的修订，通过拣选提交（`cherry-picking`）将它从被提交的源分支上移动到维护性分支上（详情可以参考第 7 章）。不过这意味着重复提交，额外的拣选提交记录有时可能会对合并行为产生不良影响。

主题分支工作流还支持用户检测特性之间的兼容性，如果有必要的话后可以修复这些冲突。可以简单地创建一个一次性的集成分支，然后将包含若干特性的主题分支集成进去，来测试它们之间的兼容性。用户甚至可以专门发布用于集成测试的这类分支（即建议式更



新或者简称 pu)，使得其他开发人员可以在开发过程中随时对它们进行校验。不过用户最好在开发文档说明中明确声明上述分支不能够作为基准进一步深入开发，因为它们每次重新创建都是来自一些不成熟的草稿。

## 主题分支工作流中某个版本的分支管理

假定我们现在正在使用 3 个节点性（集成）分支：用于维护最新版程序的 `maint` 分支，用于稳定性工作的 `master` 分支，用于功能开发的 `next` 分支。

维护者（版本项目经理）发布下一个新版本之前首先需要做的事情是确认 `master` 分支是否可以替代 `maint` 分支，即下一个版本中出现的 bug 是否都全部被修复了。用户可以通过如下命令的空白输出结果来确认这一点（详情可以参考第 2 章）：

```
$ git log master..maint
```

如果上述命令的输出结果显示还有不少未合并的提交，那么维护者需要决定如何处理它们。如果这些 bug 修复没有对程序中其他部分有任何不良影响，那么维护者就可以简单地将 `maint` 分支合并到 `master` 分支上了（因为这是将更稳定的分支合并到较不稳定的分支）。

既然现在维护者已经知道 `master` 分支是 `maint` 分支的替代品，其就可以通过给远程版本库上的 `master` 分支添加标签来创建新的版本，然后将刚才创建的标签推送到远程版本库，相关代码如下：

```
$ git tag -s -m "Foo version 1.4" v1.4 master
$ git push origin v1.4 master
```

上述命令假定项目 `Foo` 的公共版本库中的默认项是 `origin`，我们还使用了两位数表示主版本号（该命名规则遵循语义版本规范：<http://semver.org/>）。



如果维护者希望软件兼容以往的历史版本，就需要拷贝一个旧的维护性分支，因为下一个步骤是为了维护刚发布的新版程序做准备的：

```
$ git branch maint-1.3.x maint
```

然后维护者将 `maint` 分支更新到新版本，扩展该分支（需要注意的是，`maint` 分支是 `master` 分支的一个子集）：



```
$ git checkout maint
$ git merge --ff-only master
```

如果上述第二行命令无法执行,这意味着 `maint` 分支上的某些提交并没有出现在 `master` 分支上;或者更精确地说, `master` 分支并不是严格承袭自 `maint` 分支的。

因为通常我们会将特性逐个添加到 `master` 分支上,有可能部分主题分支被合并到 `next` 分支上,但是它们在合并到 `master` 分支之前就被丢弃了(或者它们还不成熟,所以还没来得及被合并)。这意味着虽然 `next` 分支是包含 `master` 分支的所有主题分支的替代品,但是 `master` 分支不一定必须是 `next` 分支的祖先。

这也是发布新版本之后扩展 `next` 分支要比扩展 `maint` 分支更复杂的原因。一种解决方案是回退并重新构造 `next` 分支:

```
$ git checkout next
$ git reset --hard master
$ git merge ai/topic_in_next_only_1...
```

用户可以使用如下代码根据未合并的主题分支对 `next` 分支进行重新构建:

```
$ git branch --no-merged next
```

在重新构建 `next` 分支之后,其他开发人员必须强制更新 `next` 分支;如果没有配置强制更新的话,那么系统将不会执行快进式操作:

```
$ git pull
From git://git.example.com/pub/scm/project
   62b553c..c2e8e4b  maint      -> origin/maint
   a9583af..c5b9256  master     -> origin/master
+  990ffec...cc831f2  next       -> origin/next (forced update)
```

这里需要注意的是强制更新 `next` 分支的操作。

## 6.2.4 Git 流——一种成功的 Git 分支模型

读者会发现更高级的主题分支工作流是基于节点分支构建的。在某些情况下,引入更多分支模型也是很有必要的,充分利用多种分支类型:节点分支、发布分支、hotfix 分支和主题分支。这类模型有时也被称为 Git 流。

这类开发模型采用两种长期节点分支,从包含集成了正在开发的最新成果的工作中把

准备发布上线的部分隔离。我们一般称这些分支是 **master** 分支（稳定性分支）和开发分支（为下一版本手机功能特性），后者还可以用于每日构建。

这两种集成分支的寿命几乎是无限的。在工作流中这些分支还伴随了一些辅助分支，例如特性分支、发布分支和 **hotfix** 分支。

每个新特性都是在主题分支上开发完成的（有时这些分支也称特性分支），其命名通常会伴随相关特性。这些分支的提取源既可以是开发分支也可以是 **master** 分支，具体的细节需要根据工作流的特性和功能特性的需要而定。当某个功能特性开发完成时，就可以使用 `--no-ff` 选项将它的主題分支合并（因此总是存在一个合并提交方便描述该特性），集成到开发分支后进行集成测试。当它们通过测试能够添加到下一版本时，将会被合并到 **master** 分支。一个主题分支的存续周期取决于其开发周期的长短，当被合并后（或者被丢弃时）就会被删除。

发布分支的用途由两部分组成。在创建时，其目标是为发布新版本做准备。其中包括文件整理，应用次要 **bug** 修复，准备版本的元数据（例如版本号、程序发布名称等）。最后需要做的是使用主题分支，准备元数据可以直接在发布分支上完成。使用发布分支使得用户可以将正在开发的特性和将要发布的主版本隔离开来，保证程序的质量。

这类分支创建的时机是稳定分支成熟或者接近成熟，即新版本待发布的状态。这种分支的每个命名都和发布的程序有关，例如 `release-1.4` 或者 `release-v1.4.x`。用户通常也会在新版本（如 `v1.4`）发布之前，根据该分支创建一些预览版程序（如 `v1.4-rc1`）。

该发布分支会一直存续到下一新版本发布为止，或者用于处理后期维护工作——修复给定分支的 **bug**（当然，通常维护工作仅限于几个最新的或者流行的版本）。对于后一种情况，它会替换其他工作流中的 **maint** 分支。

**hotfix** 分支和发布分支类似，不过它是专门用于修复预览版程序中意外出现的安全 **bug** 的。它们的命名类似 `hotfix-1.4.1`。如果对应的发布分支（维护分支）已经不存在，那么一个 **hotfix** 分支创建通常和该版本对应的标签有关。这类分支的目的主要是为了解决一些已经发布的产品中意外出现的关键 **bug**。当在这类分支上添加了一个修复提交后，相关的次级版本将会被移除（为每个这类分支）。

## 6.2.5 修复安全问题

现在假定检查属于另外一种情况。如何使用分支管理 **bug** 修复，例如一个安全问题。这里需要用到的技术和普通开发稍有不同。

如前文主题分支 workflow 所述, 虽然可以在受到 bug 影响的最稳定集成分支上直接创建一个 bug 修复提交, 不过更好的做法是针对存在的问题创建一个独立的主题分支来解决该问题。

用户可以从最原始(最稳定)的集成分支提取受到 bug 影响的内容创建一个 bug 修复分支, 也许所有分支都会受到该分支点的影响。用户将 bug 修复提交(也许会包含一组提交记录)添加到刚才创建的分支上。通过所有测试后, 用户可以简单地将该 bug 修复分支合并到受到影响的集成分支上。

这种模型还可以用于解决开发初期分支之间的冲突(依赖)。假如用户目前正在开发一些新特性(在开发分支上), 不过这些特性还不成熟。行文至此, 用户应该注意到在开发版本中遇到 bug 时如何修复它们。你希望在 bug 修复的最终状态下开展工作, 但是意识到团队其他开发人员也希望将这个 bug 修复后再继续工作。那么将 bug 修复提交记录提交到特性分支上是一个不错的方案。直接在集成分支上添加 bug 修复提交存在一个风险, 那就是在开发过程中忘记将该 bug 修复提交合并到特性分支。

解决方案是在独立的主题分支上创建一个 bug 修复提交, 然后将它们合并到正在开发的主题分支和集成测试分支上(可能还会有节点分支)。



用户可以使用类似的技术创建和管理用户定制特性的子集。用户只需为每个这类特性创建一个独立的主题分支, 然后分别将它们合并到每个客户的自定义分支上。

如果存在和安全有关的因素, 事情可能会稍微复杂一些。在解决一个严重的安全漏洞时, 用户不仅希望修复当前的版本, 可能还会考虑修复最流行的版本。为此, 用户需要为多个维护路径创建一个 hotfix 分支(从特定版本提取):

```
$ git checkout -b hotfix-1.9.x v1.9.4
```

然后, 需要将包含修复该问题的 bug 修复提交合并到刚才创建的 hotfix 分支中, 最终创建 bug 修复的版本:

```
$ git merge CVE-2014-1234
```

```
$ git tag -s -m "Project 1.9.5" v1.9.5
```



## 6.3 远程版本库上分支间的交互

如前文所述，在单个版本库中使用多个分支是非常有用的。简易的分支和合并操作使得用户能够构建强力的开发模型，可以充分利用诸如主题分支这类高级分支技术。这意味着远程版本库中也包含多个分支。因此我们的脚步不能只停留于版本库之间的交互，与此有关的内容可以参考第 5 章。

我们还必须学习在远程版本库中多个分支之间的交互。同时还要考虑本地版本库中有多少分支是和远程版本库中的分支有关联的（或者其他引用）。另外，非常重要的一点是本地版本库中的标签和其他版本库中的标签之间的关系。


理解版本库之间的交互，上述版本库中分支之间的交互，以及如何合并变更（详情可以参考第 7 章），这对于精通 Git 协作来说是至关重要的。

### 6.3.1 上游和下游

在软件开发过程中，上游一般代表某个项目的主要开发或者维护人员。我们可以将离“推举”版本库——即软件的官方源码库更近（版本库之间的层级）的版本库称为我们的上游。如果一个变更（一个补丁或者提交）被上游接受了，那么它同时也会被添加到软件的新版本中，所有处于下游的人们也会接收到它。

同样，如果本地分支中发生的变更能够完全合并和添加到远程版本库分支上的话，我们可以说远程版本库中给定分支是给定本地版本库中某个分支的上游。

#### 温馨提示：



上述远程版本库中的上游版本库和上游分支对应的给定分支，已经分别通过 `branch.<branchname>.Remote` 和 `branch.<branchname>.merge` 配置变量定义好了。上游分支可以通过 `@{upstream}` 或者快捷方式 `@{u}` 来表示。上游是在远程跟踪分支之外创建分支时设定的，用户可以使用 `git branch --set-upstream-to` 或者 `git push --set-upstream` 命令对它编辑。

上游不一定需要是远程版本库中的分支，它也可以是一个本地分支，一般来说我们称之为跟踪分支多过上游。当然，本地分支是基于另外一个本地分支时，这一特性会非常有用，



例如当一个主题分支是提取自其他主题分支而创建的（因为它包含的特性是后续分支继续工作的先决条件）。

## 6.3.2 远程跟踪分支和 refspec

在项目协作过程中，用户将会和很多版本库打交道（详情参见本章 6.2.4 “Git 流——一种成功的 Git 分支模型”）。用户打交道的每一个这种远程（公共）版本库内部都有自己的表示分支结构的方式。例如，远程版本库 `origin` 中的 `master` 分支的位置不一定必须和用户的版本库克隆中本地 `master` 分支的位置保持一致。换句话说，它们不需要在 DAG 视图中指向同一提交对象。

### 远程跟踪分支

为了能够检查集成状态，例如远程版本库 `origin` 中还有哪些变更是用户本地还没有的，或者本地工作版本库中还有哪些新增变更记录是未发布的，用户需要知道本地分支在远程版本库中对应分支的位置。这也是远程跟踪分支的主要用途——远程版本库中对应分支的引用记录（如图 6-5 所示）。

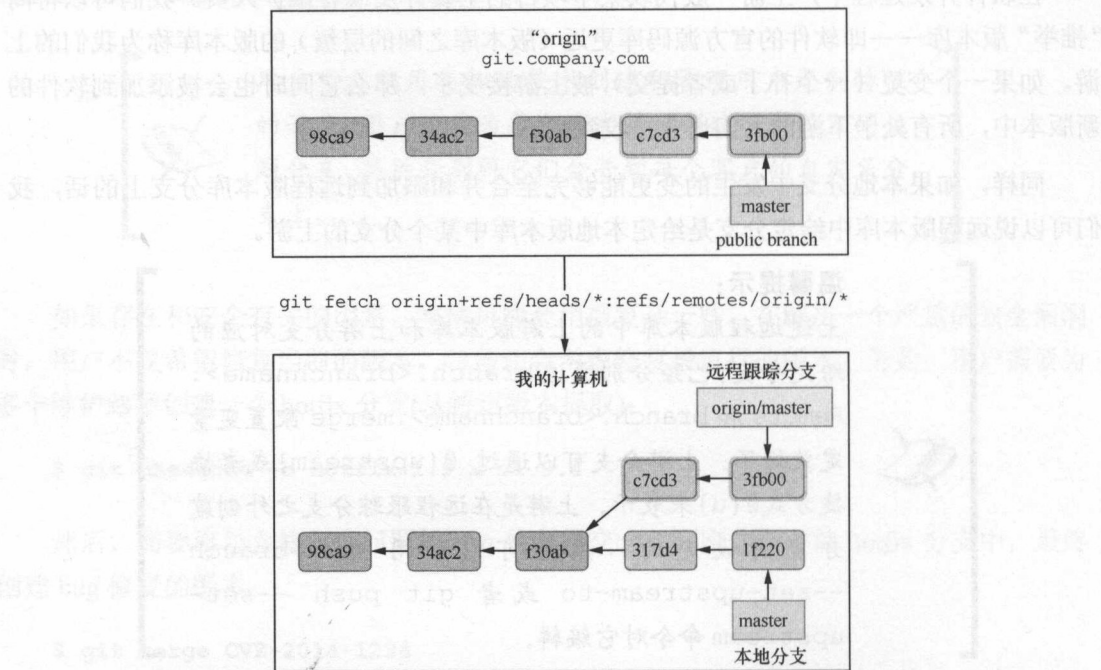


图 6-5 远程跟踪分支。在远程版本库 `origin` 中的 `master` 分支对应的远程跟踪分支是 `origin/master`（全名是 `refs/remotes/origin/master`）。`fetch` 命令中灰色的文本代表默认参数

为了跟踪记录远程版本的内部变化，远程跟踪分支是自动更新的，这意味着用户不能基于上述分支创建新的本地提交（因为在更新过程中用户会丢失这些提交）。用户需要为此创建一个本地分支。可以使用命令 `git checkout` 达到此目的，当然还需要假定用户在上述命令中指定的分支名并不存在。这个命令会在远程版本库分支<分支名>之外创建一个新的本地分支，然后为它设定相关的上游信息。

## Refspec——远程和本地分支映射规范

如第 2 章所述，本地分支所在的命名空间是 `refs/heads/`，同时给定远程版本库的远程跟踪分支所在的命名空间是 `refs/remotes/<远程版本库名称>/`。不过它们都只是默认配置。拉取（推送）操作对应的配置变量 `remote.<remote name>` 中记录了远程版本库中的分支（引用）和本地版本库中远程跟踪分支（或者其他引用）之间的映射关系。

这种映射被称为 `refspec`，它既可以是显式声明的一对一分支映射，也可以是通配符表示的映射模式。

例如 `origin` 版本库中的默认映射配置如下：

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
```

上述内容的意思是说，远程版本库 `origin` 中 `master` 分支（全名 `refs/heads/master`）中的内容将会被存放到本地版本库克隆的远程跟踪分支 `origin/master` 上（全名是 `refs/remotes/origin/master`）。模式开头的符号 `+` 是告知 Git 系统以非快进方式更新远程跟踪分支，即不继承上一版本的信息值。

这种映射可以用于远程版本库拉取操作配置，也可以当作命令行参数（只使用引用的缩写就能完全满足需要了）。如果命令行中没有对应的 `refspecs` 映射，那么该配置只会影响当前登录的用户。

### 6.3.3 fetch、pull 和 push

可以使用 `git push` 命令将变更发送（发布）到远程版本库上，同时可以使用 `git fetch` 命令将远程版本库的变更下载到本地版本库。这两个命令发送变更的方向刚好是相反的。用户需要注意的是，这 and 用户自己的本地版本库有着一个显著的差异——用户能够在键盘上输入其他命令。

这也是本地到远程版本库没有类似 `git pull` 命令获取和集成变更的原因。根本没有人能

够解决可能出现的冲突（问题在于集成变更是自动化完成的），特别是分支和标签的拉取与推送之间的差异。相关详情将会在后续章节进一步阐述。

## Pull——拉取和更新当前分支

用户经常会想将远程版本库上某个特定分支的变更集成到本地当前分支上。`pull` 命令会下载相关变更（或者执行附加相关参数的 `git fetch` 命令），然后自动将上述变更集成到当前分支上。默认情况下，它会调用 `git merge` 命令集成变更，不过用户还可以执行 `git rebase` 命令达到后者使用 `-rebase` 选项一样的效果，也可以使用 `git pull` 命令和配置选项 `pull.rebase`；或者使用 `branch.<branch name>` 为独立分支进行变基配置。

需要注意的是，如果远程版本库没有做相关配置（执行 `pull` 命令并指定 URL），Git 将会使用 `FETCH_HEAD` 引用存储获得的分支外部引用。还有，`git request-pull` 命令专门为基于 `pull` 的工作流创建发布信息或者保留变更，例如“推举”版本库工作流的变体。它会创建一个和 GitHub 合并请求等效的纯文本，因此非常适合以电子邮件的形式发送。

## 将当前分支推送到一个非裸远程版本库

一般来说，用户推送的目标版本库都是为了同步变更而创建并且是空的，即没有工作区。一个裸版本库中甚至没有当前分支的概念（`HEAD`）——没有工作树目录，因此也没有签出的分支。

不过有时用户也许会希望推送变更到非裸版本库。这是可能发生的，例如作为一种同步两个版本的方式，或者作为一种部署机制（例如一个 Web 页面或者 Web 应用）。默认情况下，Git 在服务端（用户推送的目标非裸版本库）将会拒绝当前签出分支的引用更新。这也是它不将 `HEAD` 与工作区、暂存区保持同步的原因，如果你没有注意到这一点，就肯定会感到非常迷惑。不过用户可以通过设置 `receive.denyCurrentBranch` 变量来警告或者忽略（将其由默认值改为 `refuse`）这类推送。

用户甚至可以通过和上述配置变量类似的 `updateInstead` 让 Git 更新自己的工作目录（必须清理好，保证其中不存在任何未提交的变更记录）。

一个替代性并且更灵活的解决方案是使用 `git push` 命令进行部署，在接收端配置相应的钩子。关于钩子，详情可以参考第 10 章，服务端钩子的具体应用可以参考第 11 章。

## 默认拉取 `refspec` 规范和推送模式

用户经常从公共版本库中拉取所有公开的分支项目。大部分情况也可能只是为了获取所有分支的完整更新。这也是 `git clone` 命令在 `refspec` 中设定默认拉取映射的方式，即本章



介绍的远程版本库到本地版本库的映射规范。“拉取所有”规则常见的例外是随后的 `pull` 请求。不过在这种情况下，请求中已经显式声明了版本库和分支（或者是签名标签）的状态，可以在执行 `pull` 命令时附加相应的参数：`git pull <URL> <branch>`。

另外一方面，在私人的工作版本库中，通常还有不少分支是用户不希望发布的，或者至少是用户不想马上就发布的。大部分情况下，我们只希望发布一个独立的分支，即目前用户研发完毕并通过测试的分支。当然，如果你是集成经理，你会选择发布一组经过精心筛选的分支子集，而不是单个分支。

这是拉取和推送之间另外一个重要的差异，同时也是 `Git` 不为推送操作设置默认 `refspec` 映射的原因（当然，用户也可以手工编辑该配置），不过会采用推送模式的机制来决定推送哪些内容。当然，这个配置变量只有在命令行中执行 `git push` 命令，没有显式声明推送分支状态时才起效。

#### 使用 `git push` 命令进行脱机同步

用户工作时使用机器 A 和机器 B 两台机器，其中每台机器都有自己的工作目录，常用的同步方式是在每台机器上互相使用 `git pull` 命令。不过，在特殊情况下，用户可能只能使用单向连接进行访问（例如防火墙或者间接访问的原因）。现在假定用户可以在机器 B 上执行拉取和推送操作，但是在机器 A 上只能执行拉取操作。



现在用户希望以某种方式将机器 B 上的变更推送到机器 A 上，这么做的结果会导致在机器 A 上执行拉取操作时发生混淆。为此用户需要通过 `refspec` 进行映射声明，即用户希望推送本地分支到远程跟踪分支。

```
machineB$ git push machineA:repo.git \
refs/heads/master:refs/remotes/machineB/master
```

第一个参数是类 `scp` 语法的 `URL` 地址，第 2 个参数是 `refspec` 映射地址。需要注意的是，用户完全可以通过编辑配置文件达到类似目的，这样做是为了避免重复劳动。

### 6.3.4 拉取、推送分支和标签

下一节会介绍推送模式的种类和具体用法（适合使用哪种工作流），不过首先我们需要知道 `Git` 在访问远程版本库时是如何处理标签和分支的。



因为推送和拉取并不是完全相反的操作,并且分支和标签中包含的对象也不太一样(指向一系列开发工作的分支点和特定修订的标签名),它们的行为稍有不同。

## 拉取分支

拉取分支的操作非常简单。使用默认配置的情况下, `git fetch` 命令会下载变更记录和远程跟踪分支的更新(如果有的话)。后者是根据远程版本库的拉取 `refspec` 映射完成的。当然,上述映射规则也有例外情况。这种例外情况发生在生成版本库镜像时。在这种情况下,所有的远程版本库中的引用都存放在本地版本库的同名目录下。`git clone --mirror` 命令会为 `origin` 远程版本库生成如下配置:

```
[remote "origin"]
  url = https://git.example.com/project
  fetch = +refs/*:refs/*
  mirror = true
```

引用名和它指向的对象名一起被获取了,然后被写入到 `.git/FETCH_HEAD` 文件中。这种信息已经被占用了,例如被 `git pull` 命令;如果拉取操作是通过 URL 地址而非一个远程版本库名称的话,这样做是必需的。完成上述操作后,当我们通过 URL 地址拉取变更时,被拉取的分支上没有存储远程跟踪分支信息用于集成。

用户可以使用 `git branch -r -d` 命令逐行删除远程跟踪分支配置;也可以使用 `git remote prune` 命令(在时下流行的 Git 软件中使用 `git fetch --prune` 命令)逐行删除远程版本库中已经不存在对应分支的远程跟踪分支。

## 拉取标签并自动下载标签相关引用

拉取标签的情况稍有不同。我们希望不同开发人员能够在同一分支(例如 `master` 这样的集成分支)的不同版本库中进行独立开发,这将会需要所有开发人员都可以通过某个特定标签引用同一修订版本。这也是为什么远程版本库中的分支位置信息是单独存放在远程跟踪分支的命名空间 `refs/remotes/<remote name>/*` 下,而标签有对应的镜像,每个标签都是用相同的名字存放在命名空间 `refs/tags/*` 之下。



也可以将标签的位置存放在远程版本库中;当然还可以通过相应的拉取 `refspec` 进行映射配置, Git 系统在这方面是非常灵活的。例如,可能有必要为一个子项目设置拉取配置,或者可以将标签信息存放在一个独立的命名空间中(详情可以参考第9章)。

这也是为什么默认情况下，在下载变更记录时，Git 还会拉取和本地化存储所有标签，并指向已下载的对象。用户可以使用 `--no-tags` 选项禁用自动标签关联。该选项还可以在命令行中作为参数使用，也可以通过 `remote.<remote name>.tagopt` 配置变量进行设置。

用户还可以通过 `--tags` 选项让 Git 下载所有标签，或者为标签添加合适的拉取 `refspec` 值：

```
fetch = +refs/tags/*:refs/tags/*
```

## 推送分支和标签

推送和拉取不同。推送分支通常会受到推送模式的限制。用户将一个本地分支（有可能只是单独的一个当前分支）推送到远程版本上的某个特定分支以便更新该分支上的内容，例如从本地的分支 `refs/heads/` 到远程版本中的分支 `refs/heads/`。通常它们的分支名都是一样的，不过也可以通过稍后介绍的上游具体配置来设定不同的名称。用户不需要声明完整的 `refspec` 映射名称：使用引用名称（如分支名）意味着将会推送到远程版本库上的同名引用上。如果它不存在，则会自动创建该引用。推送到 `HEAD` 意味着推送当前分支到远程版本库上的同名分支（如果没有推送到远程版本库的 `HEAD`，通常说明该引用不存在）。

通常用户推送标签需要在 `git push <remote repository> <tag>` 命令中显式声明标签名（如果偶然出现标签和分支重名的情况，即都使用了 `+refs/tags/<tag>:refs/tags/<tag>` 作为 `refspec` 映射，那么需要使用 `tag <tag>` 加以区分）。可以使用 `--tags` 选项推送所有标签（配合使用相应的 `refspec` 映射），使用 `--follow-tags` 选项启用自动标签关联功能（因为默认情况下，拉取操作是没有启用该功能的）。作为 `refspec` 映射的一种特殊情况，推送一个“空源”到远程版本库中的相关引用的含义是删除它们。`--delete` 选项是 `git push` 命令使用这种类型的 `refspec` 映射的快捷方式。例如为了删除远程版本库中的名为 `experimental` 的引用，可以使用如下代码：

```
$ git push origin :experimental
```

注意，可以在远程版本库服务器上通过 `receive.denyDeletes` 变量或钩子禁止删除引用。

## 6.3.5 推送模式应用

`git push` 命令的行为在没有指定推送目标参数和没有配置推送 `refspec` 映射的情况下，是由推送模式决定的。不同的推送模式适用的协作工作流也不尽相同。协作工作流的详情可以参考第 5 章。

## 简单推送模式——系统默认

Git 2.0 以上版本的软件采用的推送模式也称简单模式。它的设计理念非常简单：阻止推送变更到分支要强于私有变更给公共版本库带来的意外。

在这种模式下，用户总会将当然分支的变更推送到远程版本库的同名分支上。如果推送的目标版本库和拉取的源版本库是一样的（中心式工作流），那么用户需要为当前分支设置好对应的上游。上游的名称可以和分支同名。

这意味着，在中心式工作流中（拉取和推送的版本库是一样的），上游必须和当前分支（将要推送的）同名，使得上游就像获得了额外的安全保障一样。在三角工作流中，当用户推送到远程版本库中的目标和平时拉取的目标不一样时，它仍旧可以像访问当前分支那样正常运作。

这是最稳妥的方案，非常适合入门用户，这也是它被当作默认模式的原因。用户可以使用 `git config push.default simple` 命令启用该模式。

## 方便维护人员的匹配推送模式

在 Git 2.0 之前，该软件采用的默认推送模式是匹配模式。该模式在“推举”版本库工作流中对维护人员（有时也叫集成经理）最有用。但是大部分 Git 用户并非维护人员，这也是默认推送模式更换成简单模式的原因。

维护人员一般会从其他开发人员那里收到变更提交记录，收到通过 `pull` 请求或者电子邮件发送的变更补丁后，维护人员会把它们集成到主题分支中。他们也可以为自己提交的工作成果创建相应的主题分支。然后这些主题分支经过考察后，会被集成到相应的集成分支中（例如 `maint`、`master` 和 `next` 分支），合并变更记录的具体应用可以参考第 7 章。上述操作都是在维护人员的私人版本库中完成的。

公共的“推举”版本库（每个用户拉取的源版本库，详情可以参考第 5 章）中只包含长期分支（否则开发人员可能会发现自己正在开发的某个分支不知道哪天就消失了）。Git 系统自身是无法判断一个分支是短期的还是长期的。

在匹配模式下，Git 将会推送所有和远程版本库中同名的本地分支。这意味着只有已经发布过的分支才会推送到远程版本库。为了让一个新的分支支持推送，用户需要显式声明这一点，例如使用如下代码：

```
$ git push origin maint-1.4
```





需要注意的是，在这种模式下和其他模式不同，执行 `git push` 命令时不提供分支列表，它也可以一次性推送多个分支，但是不一定会推送当前分支。

为了在全局启用匹配模式，用户可以执行如下命令：

```
$ git config push.default matching
```

如果用户希望为某个特定版本库启用该模式，那么需要使用由：组成的特殊 refspec 映射格式。假定上述版本库的名称是 `origin`，现在希望对它执行非强制推送，那么可以执行如下代码：

```
$ git config remote.origin push :
```

当然，用户可以在命令行中使用推送匹配分支的 refspec 规范：

```
$ git push origin :
```

### 适用中心式工作流的上游推送模式

在中心式工作流中，只有一个公共的中心版本库供每个开发者推送访问。这个公共版本库将会只包含长期分支，通常只有 `maint` 和 `master` 分支，有时也只包含 `master` 分支。

用户永远不要直接在 `master` 工作（当然使用简单的单个主题分支是个例外），不过可以在远程跟踪分支之外为每个特性创建一个独立的主题分支：

```
$ git checkout -b feature-foo origin/master
```

在中心式工作流中，集成是分布式的：在中心公共版本库中，每个开发人员都有责任合并变更（在他们自己的主题分支上），发布最终成果到 `master` 分支。用户需要更新本地的 `master` 分支，并将主题分支合并到其中，最后推送：

```
$ git checkout master
$ git pull
$ git merge feature-foo
$ git push origin master
```

一种替代性的解决方案是对远程跟踪分支上的主题分支进行变基要好过合并它们。变



基之后，主题分支将变成远程版本库中 master 分支一个祖先，因此推送到 master 分支会很方便：

```
$ git checkout feature-foo
$ git pull --rebase
$ git push origin feature-foo:master
```

在上述两种情况下，用户可以将本地分支推送到远程版本上跟踪的对应分支（master 分支在基于合并的工作流中，特性分支在基于变基的工作流中）。在这种情况下是版本库 origin 的 master 分支。

这就是上游推送模式的工作机制：

```
$ git config push.default upstream
```

这种模式使得 Git 可以将当前分支推送到远程版本库中特定的分支上——经常集成当前分支变更记录的分支。这种分支在远程版本库上就是上游分支（用户可以通过@{upstream}访问该上游分支）。启用该模式之后，上述示例最后的命令行语句都可以得到简化：

```
$ git push
```

上游的相关信息既可以是自动创建的（在提取远程跟踪分支时），也可以是通过--track选项显式声明的。它们一般存放在配置文件中，并且使用普通的配置工具就可以进行编辑。

此外，用户还可以通过如下命令对该信息进行修改：

```
$ git branch --set-upstream-to=<branchname>
```

### 适用“推举”版本库工作流的当前推送模式

在“推举”版本库工作流中，每个开发人员都拥有自己的私人和公开版本库。在这种模式下，用户从“推举”版本库中拉取变更，推送变更到自己的公共版本库中。

在这种工作流中，用户可以创建一个新的主题分支，启动新特性的研发：

```
$ git checkout -b fix-tty-bug origin/master
```

当特性研发完成之后，用户将它们推送到自己的公共版本库上，可能还需要先执行变基操作，方便维护人员合并：

```
$ git push origin fix-tty-bug
```

这里假定用户使用 `pushurl` 配置三角工作流，推送的远程版本库名称是 `origin`。与用户为自己的公共版本库选用的是独立的远程主机时，则还需要将这里的 `origin` 名称替换成相应的推送远程版本库名称（使用独立的版本库不仅可以用于推送，还可以用来实现不同机器之间的同步功能）。

为了配置 Git 能够在 `fix-tty-bug` 分支上只执行 `git push` 命令，用户需要让 Git 系统选择当前推送模式，相关代码如下：

```
$ git config push.default current
```

该模式将会推送当前分支到接收端的同名分支。

需要注意的是，如果用户使用独立远程版本库来发布变更，则还需要配置 `remote.pushDefault` 选项，让它在执行 `git push` 命令时只支持发布推送。

## 6.4 小结

本章介绍了如何高效地使用分支进行协作开发。我们通过集成管理、发布管理、软件特性并行开发和 bug 修复等流程了解了多种分支的具体使用；了解了多种分支工作流，其中包括最常见也特别有用的主题分支工作流。上述知识能够帮助读者充分利用分支技术，自定义工作模型，以更好地为自己的项目服务。

读者还学习了如何处理单个版本库中多个分支变更记录的下载和发布。Git 为远程版本库中分支和其他信息的管理提供了很高的灵活性，其采用的机制名为 `refspec` 规范，主要是用来定义本地引用的映射：远程跟踪分支、本地分支和标签。

一般来说，拉取操作受限拉取 `refspec`，推送操作是通过可配置的推送模式管理的。不同协作工作流采用的分支发布技术也不尽相同；本章介绍了不同工作流适配的推送模式，以及如此选择的原因。

读者还学习了不少有用的技巧，其中包括如何使用“孤儿”分支技巧，在单个版本库中存储项目的 Web 页面或者生成的 HTML 帮助文档（例如 GitHub 项目页面）；了解了如何使用 `git push` 命令同步远程版本库的工作目录（例如部署 Web 应用）；还学习了在单向连接的情况下，使用推送操作达到和拉取一样的效果。

第 7 章将会介绍如何从其他分支或者开发人员那里集成变更；还会学习合并和变基，以及如何解决 Git 不能智能化处理的问题（例如如何处理多种合并冲突）。读者还会学习拣选提交和提交回退等技巧。

## 第7章

# 集成变更

上一章主要向读者介绍如何使用分支进行协作开发。

本章将会向读者介绍通过创建合并提交，或者使用变基操作，重新应用变更从多个并行开发流水线（分支）上集成变更。引入和阐释合并和变基等概念，其中包括它们的差异和具体应用。本章还会介绍多种合并冲突类型，以及如何鉴别和解决它们。

本章的主要内容包括以下几个部分。

- 合并策略和合并驱动。
- 拣选提交和提交回退。
- 补丁批量应用。
- 分支变基和提交重放。
- 文件合并算法。
- 索引中的3种状态。
- 检查和解决合并冲突。
- 使用 `git rerere` 实现记录（冲突）复用。
- 外部工具：`git-imerge`。

### 7.1 集成变更的方法

假定读者现在手头上已经有了远程跟踪分支上其他开发人员提交的变更（或者一系列补丁邮件），当然也可能包括你自己的成果，这时需要用户对它们进行整合。

又或者用户正在一个独立的主题分支上研发某个新特性，现在打算将该特性集成到长期开发分支上，以便其他开发人员能够访问。也可能用户想把一个 `bug` 修复分支集成到长期节点分支上。总之，用户希望把属于两个不同开发流水线的工作整合到一起。

Git 提供了一些整合变更的方法和这些方法的应用变体。合并操作就是其中之一，其将分属两个开发流水线的工作合并成一个双祖先的提交。另外一种方法是通过拣选提交将一个分支的工作成果拷贝到另外一个分支，即在另外一个包含相同变更集的开发流水线上创建一个新的提交（有时这么做是非常必要的）。或者用户也可以重新应用变更，通过变基将一个分支移植到另外一个分支上。接下来将介绍这些方法和它们的变体，以及相关的具体应用技巧。

大部分情况下，Git 能够自动整合变更；下一小节将会向读者介绍当系统自动合并失败或者出现合并冲突时，用户该如何处理。

### 7.1.1 合并分支

合并操作会把两个（或者更多）独立的分支整合到一块，这会把自分叉点之前的所有变更记录添加到当前分支。用户可以使用 `git merge` 命令实现这一目的：

```
$ git checkout master
$ git merge bugfix123
```

这里我们首先切换到希望合并变更记录的目标分支（本示例中是 `master` 分支），然后提供希望被合并的分支（这里是 `bugfix123` 分支）。

#### 无分叉合并——快进式和更新式

例如说用户需要为某个用户提交的 `bug` 创建一个修复提交。假定用户采纳了第 6 章介绍的主题分支工作流建议，在维护性分支 `maint` 之外创建了一个独立的 `bug` 修复提交 `bugfix123`。用户已经执行过功能测试，以便确保修复提交能够满足需要。现在用户准备将它合并，至少将它合并到 `maint` 分支中，方便其他开发人员访问。当然，也可以合并到 `master` 分支上（更稳定的分支）。后者可以用于将该修复提交部署到生产环境中。

在这种情况下，通常并没有产生真正的分叉，这意味着在维护性分支上并没产生新的提交（将要合并的目标分支），因为一个 `bug` 修复提交已经被创建了。为此，Git 系统默认情况下只会简单地将指向当前分支的指针向前移动：

```
$ git checkout maint
```



```
$ git merge i18n
Updating f41c546..3a0b90c
Fast-forward
 src/random.c | 2 ++
 1 file changed, 2 insertions(+)
```

也许用户在执行 `git pull` 命令后，当目标分支上没有新增变更记录，输出结果中会出现“Fast-forward”这样的提示。快进式合并的情况如图 7-1 所示。

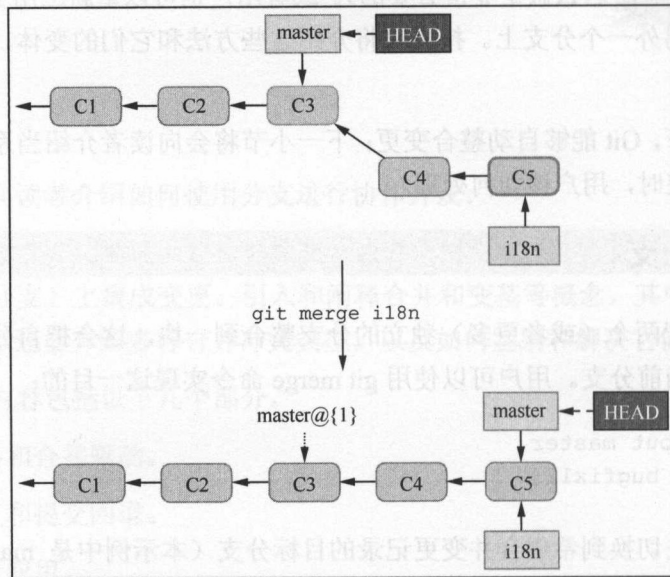


图 7-1 在合并过程中，`master` 分支的 `HEAD` 指针以快进方式移动到了 `i18n`

这种机制对于中心式和对等网络工作流（详情可以参考第 5 章）来说是非常重要的，因为快进式合并允许用户将自己的变更完全向前推送。

在某些情况下，这并不一定能够满足用户的需求。例如图 7-1 中，经过快进式合并之后，我们失去了主题分支 `i18n` 上 `C4` 和 `C5` 提交记录，并且它们是程序国际化工作的一部分。即使在这样的情况下，我们可以使用 `git merge --no-ff` 命令强制性创建一个合并提交（下一小节将会专门阐述）。该命令默认参数是 `--ff`，如果该操作执行失败，用户可以使用 `--ff-only` 选项避免系统创建一个合并提交（确保只执行快进式合并）。

另外一种情况是分支的引用是另外一个分支的祖先。理论上来说，更新式合并的情况下，被合并分支已经被包含（被合并）在当前分支了。Git 在这种情况下不需要做什么，只需要告知用户即可。

## 创建合并提交

当用户准备合并功能成熟的特性分支，而不是合并上一小节介绍的 bug 修复分支时，情况和前文所述的快进式合并不相同。开发工作通常是分开进行的。用户开始在某个主题分支上进行功能特性开发时，就将该功能和其他开发工作隔离开来了。

假定用户已经确定完成了某个功能特性的开发，并且准备将它集成到 master 分支上。为此用户需要执行一个合并操作，不过在此之前用户需要先把希望合并的分支签出，然后把希望被合并的分支名作为 git merge 命令的参数：

```
$ git checkout master
Switched to branch 'master'
$ git merge il8n
Merge made by the 'recursive' strategy.
Src/random.c | 2 ++
1 file changed, 2 insertions(+)
```

因为用户目前所分支（和将要合并的目标分支）上的顶层提交并不是目标合并分支的一个祖先或者后裔，Git 除了移动分支指针之外还必须做一些额外的工作。在这种情况下，Git 会自分叉点开始合并所有提交变更，然后将它们当作一个合并提交存储在当前分支上。因为创建提交时是基于一个以上提交（多于一个分支）的，因此它有两个父提交：第一个父提交是当前分支上一个状态的引用，第二个父提交是正在合并的分支引用。

### 注意：

如果上述操作可以自动执行的话（也就是说不存在合并冲突），Git 会向用户显示合并结果。但是事实上文本层面的合并成功并不一定代表合并结果是正确的。用户可以首先使用 git merge --no-commit 命令让 Git 系统不要执行自动合并来检查它，或者用户发现合并记录中有问题时，可以先检查相应的合并记录，然后使用 git commit --amend 命令对它们进行纠正。

与此相反，大多数其他版本控制系统并不会自动提交合并操作的结果。

Git 创建合并提交生成的内容（图 7-2 中的 M）在默认情况下（大部分情况下）是采用三路合并机制。它依次使用了被合并分支的外部应用（master: C6 和 il8n: C5）和它们二

者的祖先节点（这里是 C3，用户可以使用 `git merge-base` 命令查看它）的快照。

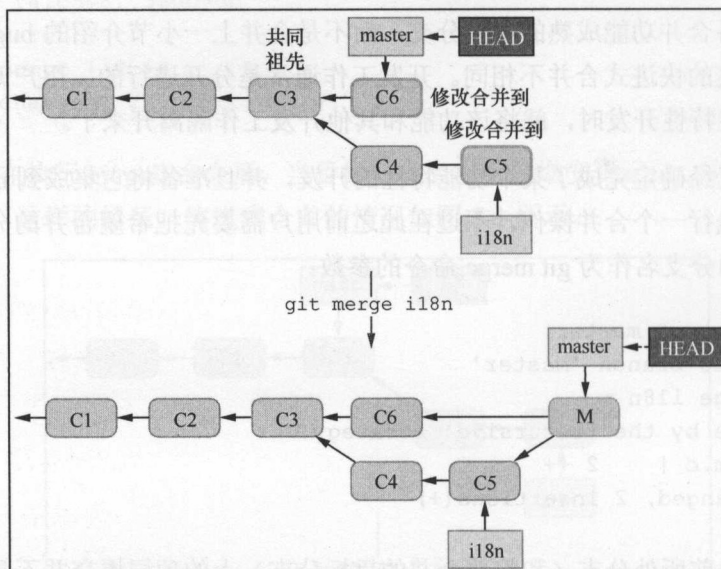


图 7-2 一个典型合并操作中用到的 3 个修订版本，以及合并操作结果

值得一提的是，Git 可以自动识别它们共同的祖先要得益于将修订记录存储在 DAG 视图和历史合并。这一机制是以往旧版本控制系统不支持的。

一个非常重要的问题是，Git 创建合并提交通常只基于 3 个修订版本：将要合并的（ours），合并目标（theirs），以及它们共同的祖先（base）。它不会检查分支的分叉部分的具体内容，因此合并速度非常快。不过正因为如此，Git 也无法识别分支上被合并的拣选提交或者回退提交，这有可能导致一些出人意料的结果（详情可以参考第 8 章）。

## 合并策略以及相关选项

从上述合并结果信息中，我们会看到它采用的是递归合并策略。合并策略是指 Git 采用的整合两个以上开发流水线上工作成果的办法，它是基于修订 DAG 视图完成的。

用户可以通过 `--strategy/ -s` 选项指定若干合并策略。默认情况下，Git 在整合两个分支时会采用递归合并策略，整合两个以上分支时会采用一个非常简单的章鱼合并策略。如果默认策略没有生效，那么用户还可以选择表决合并策略，它是一种快速而安全、功能简单的合并策略。

剩下的两种合并策略是包含特殊用途的算法。ours 合并策略可以用在用户想丢弃分支上已合并的变更，但是仍然在目标合并分支的历史记录中保留上述变更的情况。例如目的

是为了制作文档。这种策略会简单地重复当前快照（我们的修订版本）并将之作为一个合并提交。`ours` 合并策略需要注意的是，该策略是通过选项 `--strategy=ours` 或者 `-s` 指定的，不要把它和默认的合并策略相关的选项 `--strategy=recursive` `--strategy-option=ours` 或者 `-Xours` 混淆了，它们指代的含义是完全不同的。

子树合并策略可以用来处理一个主项目中从一个独立的项目迁移到一个子目录之后的后续合并。它会自动指出子项目存放的位置。这个问题和子树的概念将会在第 9 章详细介绍。

默认的递归合并策略的得名是因为它处理多个合并基底和十字形合并的方式。如果出现了有一个以上合并基底的情况（一个以上的共同祖先可以用三路合并处理），系统会以所有祖先作为合并基底创建一个合并树（冲突记录和所有提交记录），即合并操作是递归的。

当然，这些共同的祖先被合并之后也可以再次拥有一个以上的合并基底。

一些策略是支持定制的，并且包含自己的选项。用户可以在命令行中使用选项 `-X<option>`（或者 `--strategy-option=<option>`）指定合并算法的选项参数，或者通过相应的配置变量进行设置。读者将会在后续章节介绍解决合并冲突时了解到更多和合并选项有关的信息。

### 温馨提示——合并驱动

第 4 章介绍 `gitattributes` 时，也提及了不少合并驱动。这些驱动是用户自定义的，而且如果文件中存在合并冲突，会在文件级别替换默认的三路合并策略。合并策略在处理 DAG 级别的合并时与此相反，用户只能选择系统内置的选项。

### 温馨提示——签名标签和合并标签

通过第 5 章，读者已经学习了如何给自己的工作成果签名。在使用合并操作将两个开发流水线工作成果整合到一起时，用户也可以合并一个签名标签或者对一个合并提交签名（或者二者兼而有之）。对一个合并提交进行签名是执行 `git merge` 或者 `git commit` 命令时，配合 `-s` / `--gpg-sign` 选项实现的。使用后者的时机是当存在合并冲突时，或者在合并过程中使用了 `--no-commit` 选项时。

## 7.1.2 拷贝和应用变更集

合并操作就是将两个开发流水线（两个分支）上的工作整合到一起，其中包括自分叉点开始的所有变更记录。这也如第 6 章所述，如果某个提交在较不稳定的分支上（例如



master), 用户希望将它合并到较稳定的分支上时 (例如 maint), 是无法执行合并操作的。为此用户需要创建一个这类提交对象的拷贝。这种情况是可以避免的 (使用主题分支解决), 但是也有可能发生, 所以处理它们是非常必要的。

有时, 被应用的变更不一定是来自版本库 (可以是 DAG 视图上某个节点的拷贝), 但是可以是补丁的形式, 即使用 `git format-patch` 命令 (使用补丁, 并且附加注释信息) 生成的统一 diff 格式文件或者一封电子邮件。Git 内置了 `git am` 工具来处理包含变更记录补丁的大批量应用。

它们自身都是非常有用的功能, 不过理解获取变更的方法对于用户深入理解变基操作是非常帮助的。

### 拣选提交 (Cherry-pick) —— 创建变更集拷贝

用户可以使用 `cherry-pick` 命令创建一个提交 (或者一系列提交) 的拷贝。给定一系列变更记录 (通常只是一个变更记录), 它会对每个引用它的对象应用变更, 并且为每个对象创建一个新的提交。

这并不意味着源对象和拷贝对象中的快照是一样的 (即项目的状态), 后者还会包含其他变更。当然, 一般来说这些变更都是一样的 (如图 7-3 所示), 不过在某些情况下也可以不同, 例如当其中部分变更记录在更早的提交中已经存在时。

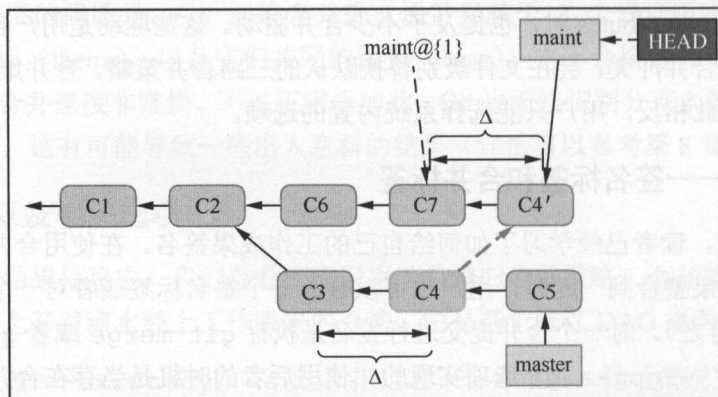


图 7-3 以拣选提交的方式将一个提交从 master 分支转移到 maint 分支。

图中靠下的虚线代表拷贝, 不过它不是一个引用

注意, 默认情况下, Git 不会存储拣选提交的来源信息。用户可以通过 `git cherry-pick -x<commit>` 命令将这些信息 (来自拣选提交对象的 sha-1 码) 附加到原生的提交对象中。这一做法仅对不存在冲突的拣选提交有效。需要谨记的是如果用户拥有访

问提交拷贝的权限，这类信息才会变得非常有用。切记不要将它用在从私有分支拷贝的提交对象上，因为其他开发人员将无法充分使用该信息。

## Revert——撤销一个提交

有时候，甚至在代码审核期间，用户会发现有些提交对象是多余的，需要将之移除（也许是事后被认为是一个不佳的设计理念，或者其中包含 bug）。如果提交对象已经被公开了，那么就不能随便将它一删了之。用户会需要消除它产生的影响，这个问题将会在第 8 章详细介绍。

“撤销一个提交”可以通过创建一个变更顺序颠倒的提交来完成，这和拣选提交有些类似，但是应用变更的顺序是颠倒的。用户还可以使用 `revert` 命令达到该目的。

该操作的名称可能会引起误解。如果用户希望撤销整个工作的所有变更，可以使用 `git reset` 命令(特殊情况下还需要指定 `--hard` 选项)。如果希望撤销对单个文件的修改，那么可以使用 `git checkout <file>` 命令。上述两个命令的具体使用已经在第 4 章详细介绍过。如图 7-4 所示，`git revert` 命令会声明一个新的提交对象，以便撤销某个历史提交对象产生的影响（一般来说包含错误的提交）。

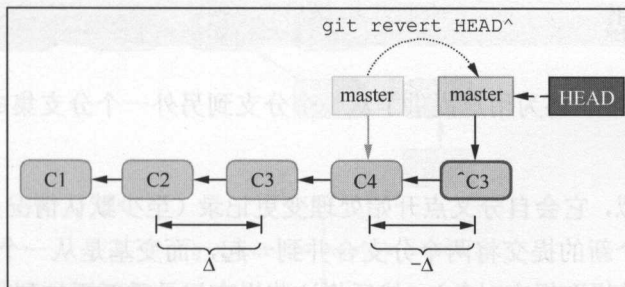


图 7-4 在 master 分支上执行 `git revert C3` 命令后的效果，创建了一个名为 `^C3` 的新提交对象

## 通过补丁应用——系列提交

在某些工作流中包含通过电子邮件（或者其他交流途径）中的补丁来交换变更。这种工作流常见于开源项目，这种方式对于新手或者零星贡献人员创建一个特定的草拟电子邮件，然后将它发送给维护人员或者邮件列表，要比配置一个公共版本库，然后发送一个 `pull` 请求简单得多。

用户使用 `git am` 命令从一个 mailbox 中应用一系列补丁（`mbox` 和 `maildir` 格式，后者只是一系列文件）。如果这些电子邮件（文件）是通过 `git format-patch` 命令生成的，那么用户可以使用 `git am --3way` 命令对它们进行三路合并，防止出现合并冲突。解决合并冲

突的问题将会在本章后续内容中详细介绍。



用户可以找到两种辅助补丁提交过程的工具，例如 GitHub 网站上的 pull 请求工具（专门为 Git 项目开发的 submitGit 的 Web 应用），以及跟踪 Web 页面补丁发送到邮件列表的工具（例如 patchwork）。

## 拣选提交和合并撤销

目前来看一切安好，不过用户希望拣选提交或者撤销一个合并提交又该如何处理呢？这类提交拥有一个以上的父提交，也就是说它们拥有一个以上的关联变更记录。这种情况下，用户必须告知 Git 系统哪些是希望拣选的变更（处理拣选提交的情况），或者使用 `-m<父提交代号>` 丢弃相关变更（处理撤销变更的情况）。

注意，撤销一个合并操作也会撤销相应的变更，但是系统不会从项目历史记录中移除合并。撤销合并操作的详情可以参考第 8 章。

### 7.1.3 分支变基

除了合并之外，Git 还为用户提供了从一个分支到另外一个分支集成变更的其他方式，其名为变基。

和合并操作类似，它会自分叉点开始处理变更记录（至少默认情况是如此）。不过合并操作是通过创建一个新的提交将两个分支合并到一起，而变基是从一个分支提取新的提交对象（自分叉点开始提取提交对象），然后将这些提交记录重新添加到其他分支的顶部。

执行合并操作时，用户首先需要切换到被合并的分支，然后使用 `merge` 命令选择希望与之合并的分支。在执行变基操作时稍有不同。首先用户需要选择一个希望变基的分支（重新应用变更记录），然后使用 `rebase` 命令选择变基的目标分支。在上述两种情况中，用户都首先需要签出分支，使其支持修改。即系统可以对该分支添加新的提交记录（合并操作会产生合并提交，变基会重新应用提交）：

```
$ git checkout i18n
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Mark messages for translation
```

用户也可以使用 `git rebase master i18n` 命令这样的快捷方式，它的效果和上述命令是一样的。在这种形式中，用户可以非常清晰地看到变基操作处理的修订区间是 `master..i18n`（这种标记在第2章已经详细介绍过），在 `master` 分支顶部重放，最终将 `i18n` 指向被重放的提交。

注意，旧版本的提交对象并没有消失，至少不是马上消失。在一定的宽限期内，用户可以通过引用日志（或者 `ORIG_HEAD`）访问它们。这意味着检查项目重放变更快照并不是那么困难，轻而易举就能查看变更集自身的状态演化历史。

## 合并和变基的优劣比较

现在已经学习了两种集成变更的方式：合并和变基。那么它们之间的区别是什么？各自又有什么优点和缺点呢？读者可以将图 7-2 和图 7-5 进行对比。第一，合并不会修改历史记录（详情可以参考第8章）。它会创建和添加一个新的提交对象（除非是快进式合并，那么它只会接近分支首部），不过提交和相关的分支都是可达的。这一点和变基有所不同。提交被重写，旧的版本和修订记录的 DAG 节点都被遗忘了。可以访问的提交对象也可能不再可达。这意味着用户最好不要对已发布的分支进行变基操作（见图 7-5）。

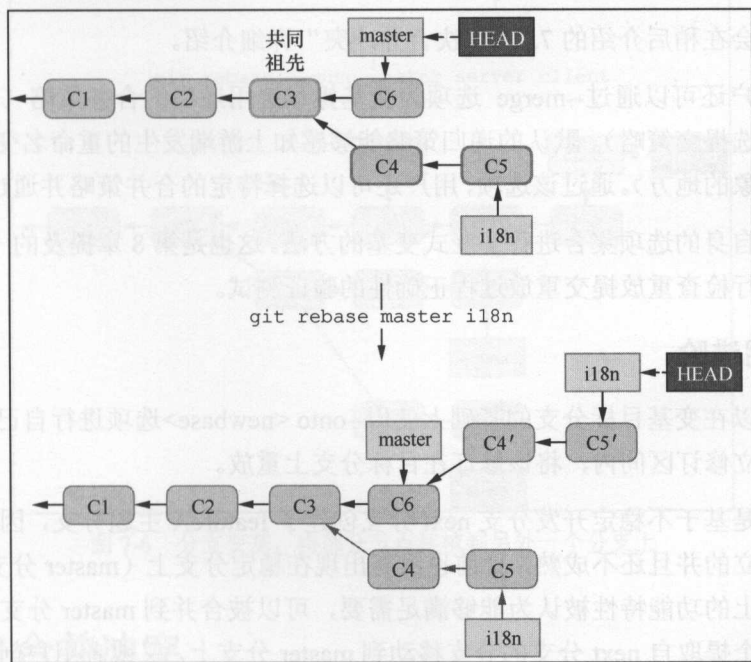


图 7-5 变基操作

第二，相关历史记录结构不一样：用户可以通过变基获得一个非常简单的线性历史记录，而执行了合并操作之后，若干系列的流水线上工作成果的提取和整合会导致历史记录



变得非常复杂。变基的历史记录更简单一些，但是丢失了变更记录在独立分支上研发分支的相关信息，用户在合并时就可以方便地获得上述信息（至少使用`--no-ff`选项）。Git 常用工具箱中甚至提供了 `git-resurrect` 这样的脚本工具，它可以用来通过合并提交中的提交信息来恢复旧的提交和已被删除的特性分支。

最后一个差异是它们的内部机制不同，变基在默认情况下重新应用提交时并不会保留合并提交的记录。用户需要使用`--preserve-merges`选项显式声明才能启用该功能。合并操作并不会改变变更历史，因此合并提交能够保留变更的原貌。

## 变基类型

上一小节已经介绍了两种拷贝或者应用变更的机制：`git cherry-pick` 命令，以及 `git format-patch` 到 `git am--3way` 的管道。这两种方式都可以用于 `git rebase` 命令重新应用变更。

默认情况下采用的是基于补丁的工作流，因为它的速度更快。使用这种类型的变基，用户在进行变基操作时可以使用一些额外的选项参数。它们实际上可以通过 `git apply` 命令实现变更集的真正重放。

这些选项会在稍后介绍的 7.2 “解决合并冲突”详细介绍。

另外，用户还可以通过`--merge`选项为变基操作选用适当的合并策略（一种针对每个提交对象的拣选提交策略）。默认的递归策略能够感知上游端发生的重命名变更（即我们放置重放提交对象的地方）。通过该选项，用户还可以选择特定的合并策略并通过选项声明它。

还有通过自身的选项集合进行交互式变基的方法。这也是第 8 章提及的一种主要工具。它可以用于执行检查重放提交重放过程正确性的验证测试。

## 变基应用进阶

用户还可以在变基目标分支的基础上使用`--onto <newbase>`选项进行自己的变基重放，即在选定的独立修订区间内，将该修订在目标分支上重放。

假定用户是基于不稳定开发分支 `next` 分支创建了 `featureA` 主题分支，因为该分支上的功能特性是独立的并且还不成熟，目前也没有出现在稳定分支上（`master` 分支）。如果主题分支 `featureA` 上的功能特性被认为能够满足需要，可以被合并到 `master` 分支上，那么用户应该希望将这个提取自 `next` 分支的分支移动到 `master` 分支上，又或者用户创建的 `server` 分支和 `client` 分支拥有共同的祖先，但是又希望 `client` 分支的独立性更明显一些。

对于第一种情况，用户可以执行 `git rebase --onto master next featureA` 命令；对于第二种情况，用户可以执行 `git rebase --onto master server client`

命令。

又或者用户只希望对分支的某个部分进行变基，那么用户可以使用 `git rebase --interactive` 命令；用户还可以使用 `use git rebase --onto <new base> <starting point> <branch>` 命令达到相同的效果。

如图 7-6 所示，用户甚至可以使用 `--root` 选项对整个分支进行变基（通常是匿名分支）。在这种情况下，用户会重放整个分支而非选定的某个子集元素。

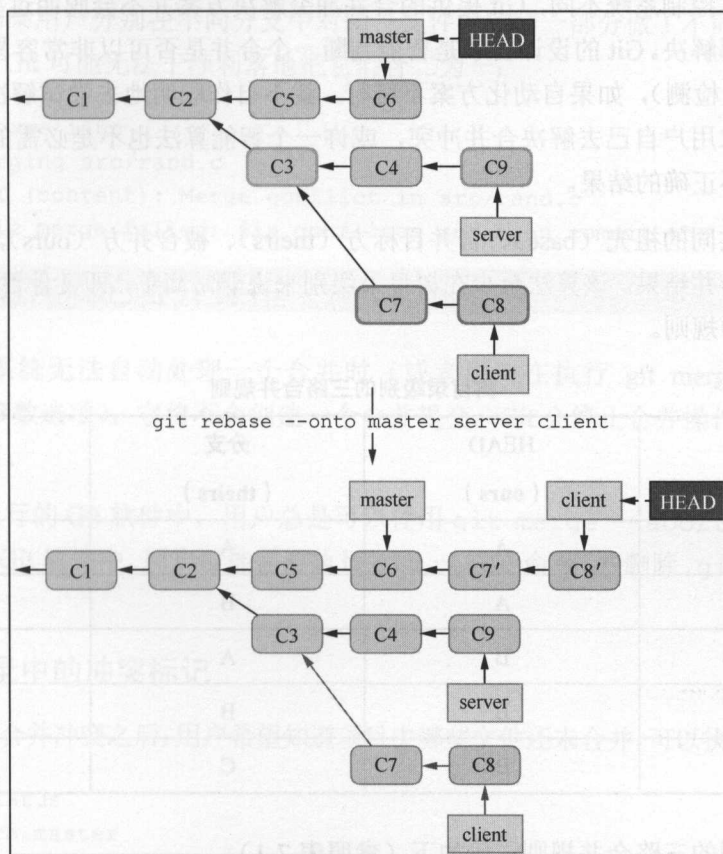


图 7-6 分支变基，将部分节点移植到另外一个分支上

## 7.2 解决合并冲突

合并操作在 Git 中是非常简单的。因为 Git 存储并可以访问完整的修订实体，它可以自动查找分支的分叉点，合并操作只和这些分叉的部分有关。这种工作甚至适用于重复合并。

因此用户可以让一个长期分支通过重复合并或者在新的变更上变基对该分支版本及时更新。不过自动整合变更的操作并不是每次都能成功执行。也有 Git 自身不能解决的问题，例如不同分支上某个文件同一区域出现了不同变更，这类问题被称为合并冲突。一般来说，这在重新应用变更时会产生问题，因此如果出现了这种情况，用户会遇到合并冲突的问题。

7.2.1 三路合并

和其他版本控制系统不同，Git 提供的合并冲突解决方案并不会聪明过头，不会尝试自动将所有问题都解决。Git 的设计哲学是智能判断一个合并是否可以非常容易地进行自动完成（例如重命名检测），如果自动化方案不可行，就不自作聪明地去尝试解决它了。最好摆脱该问题，要求用户自己去解决合并冲突，或许一个智能算法也不是必需的，这样总好过自动创建一个不正确的结果。

通过比较共同的祖先（base）、合并目标方（theirs）、被合并方（ours），Git 采用三路合并算法生成合并结果。该算法至少在树目录级别来说非常简单，即文件目录层面。表 7-1 解释了该算法的规则。

表 7-1 树目录级别的三路合并规则			
祖先 ( base )	HEAD ( ours )	分支 ( theirs )	结 果
A	A	A	A
A	A	B	B
A	B	A	B
A	B	B	B
A	B	C	merge

- 树目录级别的三路合并规则详情如下（参照表 7-1）。
- 如果一方修改变更了某个文件，选择修改版的文件。
  - 如果双方拥有相同的变更，选择修改过的版本。
  - 如果一方包含与另一方不一样的变更，那么在内容层面存在合并冲突。
- 如果存在一个以上的祖先或者某个文件的版本记录不全，那么情况会更复杂一些。但是通常了解这些规则后已经足够应付日常工作了。

如果一方的文件变更和另一方有差别（变更记录数目类型，例如一个分支上对文件重命名，它在其他分支上并不存在内容变更冲突），Git 尝试在文件内容层面执行合并，会采用预定义的合并驱动，否则就会采用内容级别的三路合并（专门针对文本文件）。

三路文件合并会检查变更是否涉及文件内的不同部分（有多少行发生了修改，这些内容变更之间都是经过差异上下文标记进行良好的区隔）。如果这些变更出现在了文件中不同的部分，Git 会自动合并它们（并告知用户哪些文件被自动合并了）。

不过，如果用户分别在不同分支中对同一文件中的同一部分做了不同修改，现在又把它们合并了，Git 可能无法干净利落地把它们合二为一：

```
$ git merge i18n
Auto-merging src/rand.c
CONFLICT (content): Merge conflict in src/rand.c
Automatic merge failed; fix conflicts and then commit the result.
```

## 7.2.2 检测失败的合并操作

当 Git 系统无法自动处理一个合并时（或者用户在执行 `git merge` 命令时指定了 `--no-commit` 参数选项），它将不会创建一个合并提交。Git 会终止合并操作，等待用户自行解决合并冲突。

在时下流行的 Git 软件中，用户总是可以使用 `git merge --abort` 命令终止合并过程。在旧版的 Git 软件中，用户可能需要执行 `git reset` 命令，并删除 `.git/MERGE_HEAD` 文件。

### 工作目录中的冲突标记

如果发生合并冲突之后，用户希望知道项目中哪些文件还未合并，可以执行 `git status` 命令：

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)
```

```
both modified:      src/rand.c
```



```
no changes added to commit (use "git add" and/or "git commit -a")
```

任何未解决冲突的内容都会被标记成未合并状态。在内容发生合并冲突的情况下，Git 会采用标准的冲突标记，将它们放在用户分支和主分支版本中存在合并冲突问题区域的周围。用户自身的文件中可能会包含下列类似内容：

```
<<<<<<< HEAD:src/rand.c
fprintf(stderr, "Usage: %s <number> [<count>]\n", argv[0]);
=====
fprintf(stderr, _("Usage: %s <number> [<count>]\n"), argv[0]);
>>>>>>> i18n:src/rand.c
```

这意味着用户版本当前分支（HEAD）对应的 `src/rand.c` 文件中有问题的程序代码块在顶部，位于<<<<<<<和=====标记之间。同时主分支版本在 `i18n` 分支上有问题的代码块在底部，位于=====和>>>>>>>标记之间。

用户需要替换整个程序块的内容来解决这个合并冲突，或者选定一方（删除另一方代码块），也可以将二者整合，例如下列代码：

```
fprintf(stderr, _("Usage: %s <number> [<count>]\n"), argv[0]);
```

为了帮助用户避免因为失误而提交一个有冲突的变更，Git 默认会检测提交的变更记录是否包含类似冲突标记的内容。如果系统发现了这类标记，则会在不使用 `--no-verify` 选项的情况下拒绝创建一个合并提交。

如果用户想检查一个共同祖先版本，以便能够解决冲突，可以切换到和冲突标记类似的 `diff3` 格式，该格式包含一个额外的代码块：

```
<<<<<<< HEAD:src/rand.c
fprintf(stderr, "Usage: %s <number> [<count>]\n", argv[0]);
|||||
fprintf(stderr, "Usage: %s <number> [<count>]\n", argv[0]);
=====
fprintf(stderr, _("Usage: %s <number> [<count>]\n"), argv[0]);
>>>>>>> i18n:src/rand.c
```

可以通过如下命令重新对每个文件进行检查来单独替换合并冲突标记：

```
$ git checkout --conflict=diff3 src/rand.c
```

如果用户比较偏爱这种格式，那么可以将它设置为默认的合并冲突格式，即将 `merge.conflictStyle` 的值设为 `diff3`（从默认的合并格式）。

### 索引中的 3 种状态

不过，Git 是如何跟踪记录哪些文件已经被合并，哪些没有呢？在工作目录文件中的冲突标记显然是不能满足需要的。有时文件中一些合法的内容和提交标记类似（例如用于测试合并的测试文件内容，或者 `AsciiDoc` 格式的文件），而且存在比冲突内容还要多的冲突类型。那么 Git 是如何处理这些内容的？例如两方以不同方式对某个文件进行了重命名，或者一方修改了文件，而另一方将该文件删除了。

事实证明，这是暂存区中的提交（这种情况下是一个合并提交）的另外一种使用方法，用户有时也将暂存区称为索引。在出现合并冲突的情况下，Git 会将所有冲突文件版本以一定的状态进行存储；每种状态都对应了一个数字代码。状态 1 是共同的祖先（base）；状态 2 是自 `HEAD` 起的将要被合并的修订版本，即当前分支（ours）；状态 3 来自 `MERGE_HEAD`，即目标合并分支（theirs）。

用户可以通过底层命令 `git ls-files --unmerged` 查看这些未合并文件的状态（或者使用 `git ls-files -stage` 命令查看所有文件的状态）：

```
$ git ls-files --unmerged
100755 ac51efdc3df4f4fd318d1a02ad05331d8e2c9111 1 src/rand.c
100755 36c06c8752c78d2aaf89571132f3bf7841a7b5c3 2 src/rand.c
100755 e85207e04dfdd50b0a1e9febbc67fd837c44a1cd 3 src/rand.c
```

用户可以通过：`<stage number>:<pathname>` 这样的声明语法访问每个版本，例如当用户想查看 `src/rand.c` 文件的一个共同祖先的版本时，可以执行如下命令：

```
$ git show :1:src/rand.c
```

如果不存在合并冲突，文件在索引中是以状态 0 标记的。

### 差异比较——组合 diff 格式

用户可以使用 `status` 命令查看哪些文件还未被合并，冲突标记能够出色地将有冲突的部分展示出来。如果想在着手解决合并冲突之前查看冲突记录，以及如何查看冲突解决的过程，该怎么做呢？答案是 `git diff` 命令。

对于合并操作有一点需要谨记，即使在合并过程中，Git 都将会以名为组合 diff 的格式

显示相关的信息。它的信息显示类似下面的内容(主要是提示合并过程中出现的冲突文件):

```
$ git diff
diff --cc src/rand.c
index 293c8fc,4b87d29..0000000
--- a/src/rand.c
+++ b/src/rand.c
@@@ -14,16 -14,13 +14,26 @@@ int main(int argc, char *argv[])
    return EXIT_FAILURE;
}

++<<<<<< HEAD
+ int max = atoi(argv[1]);
+ if (max > RAND_MAX) {
+     fprintf(stderr, "Cannot handle <number> larger than %d (%d)\n",
+         RAND_MAX, max);
+     return EXIT_FAILURE;
+ } else if (max < 2) {
+     fprintf(stderr, "<number> cannot be smaller than %d (%d)\n",
+         2, max);
+     return EXIT_FAILURE;
+ }
++=====
+ char *endptr = NULL;
+ long int val = strtol(argv[1], &endptr, 10);
+ if (*endptr) {
+     fprintf(stderr, "Invalid argument(s)\n");
+     return EXIT_FAILURE;
+ }
+ int max = (int) val;
++>>>>>> 8c4ceca59d7402fb24a672c624b7ad816cf04e08

srand(time(NULL));
int result = random_int(max)
```

可以看到它和第3章介绍的统一 diff 格式有一些细微的差异。首先,它在首部使用 diff --cc 声明采用的是紧凑的组合格式(如果用户使用的是 git diff -c 命令,也可以采用 diff --combined 代替该声明)。扩展首部行,例如索引 293c8fc, 4b87d29..0000000, 这说明存在一个以上的源码版本。@@@ -14, 16 -14, 13 +14, 26 @@@作为首部主干被修改过了(和普通的补丁不同),这是为了防止用户将组合 diff 格式当作统一 diff 格式使用,例

如通过 `patch -p1` 这样的命令。

`diff` 命令执行结果中的每一行都是以两个或者多个字符前缀开头的（两个合并分支的情况通常是以两个字符开头）：第 1 个字符告诉用户所在行与最终结果比较的第一原像（用户分支上）的状态，第 2 个字符告知用户其他原像的信息（主干分支）等。例如，`++` 代表该行并没有出现在任何被合并的版本中（这里的示例中，用户可以在包含冲突标记的行中找到它）。

差异比较对于处理合并冲突问题来说是非常有帮助的。

为了将当前分支的版本（被合并分支）和最终结果中的版本（工作目录中的当前状态）进行比较，即比较用户自己的版本，那么可以使用 `git diff --ours` 命令。类似的情况下，也适用于目标合并版本（`theirs`）以及共同的祖先版本（`base`）。

### git log——merge 命令

有时我们需要更多的上下文信息来决定选择哪个版本或者以其他方式解决合并冲突。一种方法是回顾少量历史记录，记住两个开发流水线被合并时涉及相同区域代码的原因。

为了获得分叉点之后的所有提交列表，我们可以使用第 2 章介绍的三点修饰符语法，并使用 `--left-right` 选项让 Git 显示给定提交对象属于哪一方：

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
```

可以对上述命令进一步简化，并将查询范围限制在有冲突的文件之内，即执行 `git log` 命令时添加 `--merge` 选项：

```
$ git log --oneline --left-right --merge
```

这一点对于帮助用户快速获取了解某些冲突产生的原因，以及更智能地解决相关问题来说是非常有帮助的。

## 7.2.3 避免合并冲突

Git 系统偏向于以明确的方式放弃处理不能自动合并的操作，而不是尝试使用某些精巧的合并算法。有不少工具和选项可以帮助 Git 系统避免遇到合并冲突。

### 有用的合并选项

在合并分支中，有可能遇到的一个问题是它们使用了不同的尾行转换规范或者清理过



过滤器（详情可以参考第4章）。出现这种情况可能是在一个分支添加了这类配置（修改 `gitattributes` 文件），而另外一个分支则没有。在尾行字符转换配置发生变更时，用户可能会发现很多伪造的变更记录，即行与行之间的唯一差异就是 EOL 字符不同。对于上述两种情况，在进行三路合并时，用户可以执行一个虚拟的签出，然后对所有文件的 3 种状态一起执行签入。这是通过对递归合并策略制定规范化选项参数来解决问题的（`git merge -Xnormalize`）。当然，这也可以通过它的名称那样规范化所有行间字符，然后使它们在 3 种状态下都保持一致。

修改文件行尾可能会被当作和空格冲突相关的一部分。这种情况非常容易辨别，其中一方的每一行空格都被移除了，而另一方则再次添加了空格。`git diff --ignore-whitespace` 命令会显示一个更易于管理的冲突列表（即使该冲突已经被解决了）。如果用户发现在一个合并中存在大量的空格问题，那么可以终止和重做它，不过这次要在相关命令中添加 `-Xignore-all-space` 或者 `-Xignore-space-change` 选项参数。注意，一行中和其他变更混在一起的空格是不会被忽略的。

有时错误的合并是由无关紧要的行匹配导致的（例如不同功能之间的花括号）。用户可以通过 `-Xpatience` 或者 `-Xdiff-algorithm=patience` 选项，让 Git 采用更耗时的差异比较算法将版本之间的差异最小化。

如果问题是由错误的重命名检测引起的，那么可以通过 `-Xrename-threshold=<n>` 选项调节重命名阈值。

## 20! Rerere（历史方案复用）

**Rerere**（历史方案复用）功能是一个不那么容易发觉的特性。顾名思义，它会让 Git 逐个主干地记录解决其中冲突的过程，以便系统下次遇到同类问题时可以自动解决它。不过需要注意的是，Git 会停在解决冲突的地方，并且不会自动提交该基于历史方案的 `rerere`，即使它可以完美地解决该冲突（如果表面看起来是正确的）。

这个功能在很多场景中都非常有用。例如用户希望将一个长期分支完整地合并到头，但是又不想创建过渡合并提交。在这种情况下，用户可以做实验性的合并（先合并，然后删除该合并），将合并冲突的过程信息保存到 `rerere` 缓存中。通过这种技术，最终合并将会变得非常简单，因为大部分冲突已经根据早期的历史方案被解决了。

另外一个可以充分利用 `rerere` 缓存的场景是：当用户将一大堆主题分支合并到一个支持测试的长期分支上时。如果某个分支的集成测试失败了，但是用户又不想把时间浪费在解决一个合并冲突上。又或者用户认为使用变基比合并效果更好。`rerere` 机制允许用户将合

并历史方案迁移到变基方案中。

为了启用这个功能，只需要简单地将 `rerere.enabled` 的属性设置为 `true`，或者创建 `.git/rr-cache` 文件即可。

## 7.2.4 处理合并冲突

假定 Git 不能完美地进行自动化合并，现在需要用户亲自处理合并冲突，以便能够创建一个新的合并提交。那么用户该怎么做呢？

### 终止合并

首先，简要介绍一下如何处理这种情况。如果用户还没有准备好处理冲突或者对于如何解决它们所知甚少，那么可以使用 `git merge --abort` 命令，将该合并操作自起点开始丢弃。

该命令尝试将状态重置到用户执行合并操作之前。如果用户不是从一个干净的状态开始执行合并操作的，那么可能就不能采用上述方法了。因此在许可的范围内，在执行一个合并操作之前，最好将变更暂存。

### ours 和 theirs 的版本选择

有时，在有冲突的情况下，选择一个版本就可以解决了。如果用户想用这种方法来解决所有冲突，强制所有主干分支通过对 `ours`（用户主题分支）和 `theirs`（长期分支）版本的选择来解决冲突，可以使用选项 `-Xours`（或者 `-Xtheirs`）或者递归合并策略。需要注意的是，选项 `-Xours`（合并选项）和 `ours` 的选项 `-s`（合并策略）是不一样的。后者会创建一个伪合并，合并的内容和 `ours` 版本是一样的，而非只为冲突文件选择 `ours` 版本。

如果只希望对某些特定文件进行上述操作，那么可以使用 `git checkout --ours / --theirs` 命令，重新签出 `ours` 或者 `theirs` 版的文件。

用户可以分别使用 `git show :1:file`、`:2:file`、`:3:file` 命令，查看相关文件 `base` 版、`ours` 版和 `theirs` 版的详细内容。

### 脚本化修复——手动合并文件

有几种变更是 Git 不能自动处理的，不过可以用脚本调用来弥补。如果用户可以首先将“`ours`”“`theirs`”和“`base`”的版本进行转换，那么合并操作就可以自动化完成，至少能够更容易一些。重新规范版本库中发生变更后的文件签出和存储的方式（`co1` 和清理/冗余

信息过滤器), 使用内置的选项处理空格引起的变更。另外还有非内置的示例是修改文件编码格式, 以及其他可以通过脚本进行指定的配置项, 例如重命名变量。

为了执行一个脚本化合并, 用户首先需要提取冲突文件每种版本的副本, 可以通过 `git show` 命令, 搭配:`<stage>:<file>` 这样的语法指定参数:

```
$ git show :1:src/rand.c >src/rand.common.c
$ git show :2:src/rand.c >src/rand.ours.c
$ git show :3:src/rand.c >src/rand.theirs.c
```

现在用户已经有了工作区中全部的 3 种状态的文件, 然后就可以单独对它们的每个版本进行修复了, 例如使用 `dos2unix` 或者 `iconv` 等工具。用户可以通过如下代码对文件的内容进行重新合并:

```
$ git merge-file -p \
    rand.ours.c rand.common.c rand.theirs.c >rand.c
```

## 使用图形化合并工具

如果希望使用一款图像界面工具来帮助自己解决合并冲突, 那么可以执行 `git mergetool` 命令, 它会启动一个可视化合并工具, 引导用户处理所有合并冲突。现在有大量的预配置图形化合并助手工具供用户选择。

用户可以配置 `merge.tool` 属性, 选择自己喜欢的合并工具。如果用户没有这么做, 那么 Git 系统会根据操作系统和桌面环境依次尝试调用可用的图形化工具。

用户还可以根据偏好定制自己用得顺手的工具。

## 将文件标记为已解决并最终合并

如前所述, 如果一个文件中存在合并冲突, 它在索引中会包含 3 种状态。为了将一个文件标记为已解决的, 用户需要将文件的内容转换为 `stge 0`。这可以通过简单地执行 `git add <file>` 命令实现。当所有冲突都解决后, 用户需要执行 `git commit` 命令完成最终的合并提交 (或者可以跳过将每个文件标记为已解决的状态, 只执行 `git commit -a` 命令)。

默认的提交说明信息就是被合并内容的信息摘要, 其中还包括一组合并冲突列表, 并且默认会添加合并目标分支的短日志。该日志可以通过 `--log` 选项和 `merge.log` 配置变量进行控制。



## 解决变基冲突

当应用一个补丁或者一系列补丁、拣选提交或者提交回退，又或者分支变基等操作碰到问题时，Git 将会保守地使用三路合并算法。关于如何解决上述情况下的合并冲突，前面已经讲过。

不过，对于这些方法来说，例如变基、应用 mailbox 补丁或者拣选一系列的提交记录，它们是通过逐个阶段进行处理的（一个操作序列），它们还有其他问题，即如果在这样的一个独立阶段中出现了合并冲突，该怎么处理呢？有 3 种方案可供用户选择。用户可以解决该冲突，然后使用选项 `--continue`，继续执行相关操作（或者执行 `git am` 命令，并搭配 `--resolved` 选项）。用户也可以使用选项 `--abort` 终止该操作，并将当前分支的 HEAD 重置到原生分支。最后，用户也可以使用 `--skip` 选项，丢弃一个版本，也许因为它已经出现在了上游分支上，用户可以在重放过程中将它丢弃。

### git-imerge——git 的增量合并与变基

变基和分支都存在它们自身的缺点。对于合并，用户需要以要么全有要么全无的方式解决一个严重的合并冲突（即使使用测试合并和 `rerere` 来保留最新的方案能够有所帮助）。几乎不存在保存部分已完成合并记录并能够测试它的方式；`git stash` 命令可以帮上忙，但是它也并非完美的解决方案。

换句话说，变基是以按部就班的方式完成的。但是它对协作开发不太友好，用户最好不要对已发布的历史记录进行变基。用户可以中断变基，但是它会让你处于一个奇异的状态（在一个匿名分支上）。这也是第三方工具 `git imerge` 诞生的原因。它可以使用小的步骤但将冲突结对表示。它以这种方式记录所有中间状态的合并以方便团队共享，因此一个人可以启动合并，另外一个人可以将它最终完成。最终的方案可以另存为普通合并、普通变基和带历史记录变基。

## 7.3 小结

本章向读者介绍了如何高效地将两个开发流水线合二为一，以及如何自分叉点开始的所有提交对象集成它们。

首先我们学习了整合变更的几种方法：合并、拣选提交和变基。这一部分主要将精力集中在了如何在更高层次应用它们的功能，即在修订 DAG 视图层面。读者学习了合并和变基的基本工作原理，以及它们之间的异同。变基的某些用法非常有趣，例如将一个长期



分支上的主题分支移植到另外一个长期分支。

然后，读者学习了在 Git 不能自动化整合变更时该怎么做，即出现合并冲突时该如何解决。这一过程比较重要的部分是理解三路合并算法的工作原理，以及索引和工作区在出现冲突时受到的影响。读者现在已经知道如何检查失败的合并操作，以及如何查看历史建议方案，如何避免出现合并冲突，最终解决合并冲突并把它们标记为已解决状态。

第 8 章将会向读者阐释如何重写修订历史，以及保持其简洁性的原因（和这么做的意义）。交互式变基是重写修订历史的工具之一，本章介绍的普通变基是它的一个近亲。读者将会学习多种重写提交的方法：如何对它们重排；如果它们体积太大，如何分割它们；如何将修复补丁插入提交以确保其正确性；如何从历史记录中删除一个文件。该章还介绍了在用户不重写修订历史的情况下，可以做的事情有哪些（充分理解重写已发布修订历史记录被认为是一种糟糕的做法的原因），不过用户仍然需要对它进行纠正——使用置换和笔记机制。同时我们还会在其他应用中讨论这些机制。

## 第 8 章

# 历史记录管理

上一章主要介绍的内容是如何整合不同开发人员（详情可以参考第 5 章）或者某个独立的特性分支（详情可以参考第 6 章）提交的工作成果。变基作为其中涉及的技术之一，能够帮助用户将一个分支合并到一个更佳的状态。那么是否可以通过变基修改提交记录方便代码审核，使得功能开发特性流程更简洁一些呢？在无法重写历史记录的情况下，如何修复错误呢？

本章将会回答上述所有问题。同时还会解释用户可能希望历史记录保持整洁的原因，以及实际工作中的最佳实践的时机和具体措施。这里还会有条不紊地向读者介绍提交的重排、插入和分割等方法的详细步骤。本章还会介绍如何对大型项目历史记录重写（例如自其他 VCS 导入后的清理工作），以及当用户无法重写版本历史记录时的解决方案（还原、置换和笔记等方法的具体应用）。

为了真正理解本章涉及的某些主题并且精于实践，用户首先需要学习本章之初阐释的一些 Git 内部机制的基础知识。

本章的主题包括以下几个方面。

- Git 版本库对象模型基础。
- 用户不应该重写已发布的版本历史的原因，以及如何还原它们。
- 交互式变基：重排、插入、分割和测试提交对象。
- 修订还原，合并还原，合并还原之后的重新合并。
- 大型脚本化重写版本历史。
- 在不重写的前提下，使用移植和置换机制编辑版本历史记录。

- 为包含笔记的对象附加额外信息。

## 8.1 Git 内部机制简介

为了真正理解和掌握本章介绍的一些方法，用户至少需要理解基本的 Git 内部原理。此外，读者还需要知道 Git 是如何存储和修订有关的信息的。

同时某些读者可能还需要了解如何编辑这类数据，以及通过脚本操作这类数据。Git 提供了一组底层命令专门用于编写脚本，它们是面向用户的高层命令的有益补充。这些命令不但灵活而且功能强大，不过可能并不是那么人性化。了解这类脚本接口知识有助于帮助用户通过钩子管理 Git 版本库，详情可以参考第 11 章。

### 8.1.1 Git 对象

第 2 章已经向读者介绍过，Git 表示版本历史记录的方式是通过修订的有向无环图 (DAG)，每个修订用图中的一个节点表示，即提交对象。每个提交通过一个 SHA-1 标识符进行标记。用户可以通过这种标识符引用任何给定版本（用全名或者一个不会引起歧义的简称）。

提交对象中包含了修订的元数据，指向了 0 个或者多个父提交，而且修订代表项目文件的快照。修订元数据包含作者信息（何人、何时创建的该变更记录）、提交者信息（何人、何时创建的该提交对象），当然还有提交对象的备注信息。

Git 表示给定修订的项目文件快照的方式很有意思。Git 使用树对象表示文件目录，用二进制大对象 (BLOB) 表示文件内容。除了提交、树对象、二进制对象以外，还有用于表示附注和签名标签的标签对象。

每个对象会根据其内容，或者更精确地说是对象的大小和类型以及内容，通过 SHA-1 哈希函数进行标识。这种基于内容的标识符不需要依赖一个中央命名服务。基于这个事实，每个分布式版本库的同一项目将会使用相同的标识符，并且用户无须担心出现命名冲突：

```
# calculate SHA-1 identifier of blob object with Git
$ printf "foo" | git hash-object -t blob -stdin
# calculate SHA-1 identifier of blob object by hand
$ printf "blob 3\0foo" | shasum
```

我们可以将 Git 版本库称为内容编址对象数据库。当然，并不能一概而论，还有引用

(分支和标签) 和多种配置信息以及其他的一些东西。

接下来将进一步自下而上介绍 Git 对象的细节。用户可以使用底层命令 `git cat-file` 查看这些对象：

- **Blob:** 这些对象用于存储文件内容的给定版本。可以使用底层命令 `git hash-object -w` 创建一个这样的对象。注意，如果不同修订版本包含的文件内容是一样的，那么得益于内容编址技术，它只会存储一次：

```
$ git cat-file blob HEAD:COPYRIGHT
Copyright (c) 2014 Company
All Rights Reserved
```

- **树:** 这些对象用于表示文件目录。每个树对象是一个根据文件名排序的实体列表。每个实体由复合权限和类型、文件或者目录名，给定路径的已连接对象的一个链接（即 SHA-1 标识符），或者树对象（表示子目录）、blob 对象（表示文件内容），又或者只是提交对象（表示子模块）等元素构成。注意，如果不同修订之间包含一个子目录的相同内容，那么得益于内容编址技术，它只会存储一次：

```
$ git cat-file -p HEAD^{tree}
100644 blob 862aafd... COPYRIGHT
100644 blob 25c3d1b... Makefile
100644 blob bdf2c76... README
040000 tree 7e44d2e... src
```

注意，上述内容实际的输出包含完整 40 个字符的 SHA-1 标识符，并不是和上述示例中那样采用简称。用户可以在索引（可以使用 `git update-index` 命令创建索引）之外使用 `git write-tree` 命令创建一个树对象。

- **提交:** 这类对象表示修订版本。每个提交由一组包含 0 个或多个父提交的首部（键-值对）构成，实际上包含一系列链接的树对象表示版本库内容的快照——项目的顶层目录。用户可以使用底层命令 `git commit-tree` 或者只是简单地使用 `git commit` 命令创建一个包含给定树对象的提交，将它作为一个修订的快照：

```
$ git cat-file -p HEAD
tree 752f12f08996b3c0352a189c5eed7cd7b32f42c7
parent cbb91914f7799cc8aed00baf2983449f2d806686
parent bb71a804f9686c4bada861b3fcd3cfb5600d2a47
author Joe Hacker <joe@example.com> 1401584917 +0200
committer Bob Developer <bob@example.com> 1401584917 +0200
```



```
Merge remote branch 'origin/multiple'
```

- **标签：**这些对象表示附注标签，签名标签属于特例。标签对象还包含一系列的首部信息（此外还指向了一个标签对象）和一个标签信息。用户可以使用底层命令 `git mktag` 或者简单地使用 `git tag` 命令创建一个标签：

```
$ git cat-file tag v0.2
object 5d2584867fe4e94ab7d211a206bc0bc3804d37a9
type commit
tag v0.2
tagger John Tagger <john@example.com> 1401585007 +0200
```

```
random v0.2
```



Git 对作者、提交者和标签日期采用的内部格式是 `<unix timestamp> <timezone offset>`。UNIX timestamp (POSIX 时间)是继 UNIX 纪元之后的秒数，即国际标准时间(UTC)1970 年 1 月 1 日 00:00:00(1970-01-01T00:00:00Z)、星期四，并且不计算闰秒。这是指当事件发生时，用户可以使用"%s"打印 UNIX 时间戳，并可以使用日期选项 `--date="@<timestamp>"` 将它们转换成其他格式。

时区偏移是 HHMM (小时和分钟) 格式对 UTC 做正负偏移的方式。例如 CET (欧洲中部时间，比 UTC 早 2 个小时) 的时区偏移值是 +0200。它可以用来确定某个时间的当地时间。

某些 Git 命令可以处理任意类型的对象。例如用户可以给任意类型的对象添加标签，不仅限于提交对象。此外，用户还可以给一个 blob 对象添加标签，以便可以在版本库中保留某些不相关的数据，并且让它在每个克隆中也生效。公钥就是这样一种数据。本章后续将要介绍的笔记和置换也可以处理任意类型的对象。

## 8.1.2 Git 的底层命令和高层命令

Git 是采用自下而上的方式研发的。这意味着它的开发是从基本程序开始，逐步完善和添加功能的。很多面向用户的命令在编译之初是作为 shell 脚本调用底层命令执行相关操作的，因此我们很容易区分这两类 Git 命令。

广为熟知的类型是所谓的高层命令，它是高级的、面向用户的命令（“porcelain”是对引擎级管道命令的戏称）。这类命令的输出结果是专门为最终用户准备的。这意味着它的输出结果可以加以修饰，变得更人性化。因此，它的输出结果在不同版本的 Git 程序中可能不尽相同。由于 Git 版本的差异，用户应该有足够的辨别能力知道发生了什么，例如额外的信息，或者被修改过的措辞、格式变更等。

这种情况和可能需要用户编写的脚本不同（指本章介绍的脚本，例如使用 `git filter-branch` 命令进行脚本化重写过程中的一部分）。这里用户需要保证输出内容不发生变更。至少对于多次使用的脚本是如此（例如作为钩子、`gitattribute` 驱动、辅助助手等）。用户会发现选项 `--porcelain` 出现的频率很高，它的目的是为了确保命令的输出结果不发生变更。对于其他命令，解决方案是指定完整的格式。另外，用户还可以在脚本中使用底层命令，即被称为管道化的命令。这些命令默认情况下并不是那么人性化，更不用说智能感知功能了。它们的输出结果并不会受限于 Git 配置；能够通过 Git 配置文件进行配置的底层命令非常少。

帮助手册 `git(1)` 中分别列出了所有底层命令和高层命令。第 4 章首次谈到了底层命令，并且该命令没有对应的面向用户的高层命令时，已经介绍过底层命令和高层命令之间的区别。

## 8.1.3 Git 环境变量

Git 采用了若干 shell 环境变量来决定自身的行为。对于面向用户的高层命令来说是 shell 脚本，它们被用来将数据传递到底层命令执行相关工作，此外还使用标准的输入（管道）和命令行参数。

个别情况下，用户可以很方便地知道这些环境变量是什么，以及通过它们让 Git 系统完成用户的指令的具体过程。这在后续章节中介绍 `git filter-branch`（特别是选项 `--env-filter`）命令进行脚本化历史重写时有非常直观的过程演绎。

环境变量的优先级在 Git 配置和命令行参数之间：环境变量优于配置信息，命令行参数

的优先级高于环境变量。除了退化的非 Git 声明的环境变量之外，例如 `PAGER` 和 `EDITOR`，即选用最低优先级，并且可以被诸如 `core.pager` 和 `core.editor` 这样的配置变量覆盖。

接下来的内容并不包含 Git 使用的环境变量的完整列表，只会选择一组非常有用并且和本章主题相关的进行介绍。

## 环境变量对全局行为的影响

某些 Git 基本的行为大体上会依赖环境变量（搜索路径和调用的外部程序）。

`GIT_EXEC_PATH` 决定了 Git 核心程序的安装位置。用户可以使用 `git --exec-path` 命令检查和配置当前的设置。默认值是在程序安装时配置的。

Git 查找配置文件时也会受到环境变量的影响。用户特定配置文件（有时也称全局配置文件）可以在 `$XDG_CONFIG_HOME/git/config`（如果 `XDG_CONFIG_HOME` 未设置或者为空的话，目录为 `$HOME/.config/git/config`）或者 `$HOME/.gitconfig` 中找到，后者文件中的值遵循优先级规则。用户可以在一个完全便携式 Git 安装包中的 `shell` 属性中的 `HOME` 和 `XDG_CONFIG_HOME` 重写这些变量的值（当然还有其他若干依赖项）。

系统级配置文件的路径是在安装 Git 时设定的（同时可以在 `/etc/gitconfig` 中配置），不过用户可以通过设置环境变量 `GIT_CONFIG_NOSYSTEM` 跳过该配置的读取。如果用户系统配置对 Git 命令有干扰的话，这将是非常有用的。或者用户可以通过 Git 的环境变量 `GIT_CONFIG` 指定独立的配置文件（`git config` 命令和选项 `--file` 的作用是一样的）。

如果标准输出界面是用户登录终端的话，通过设置环境变量 `GIT_PAGER` 可以让程序在命令行以多页面的形式显示结果。如果没有设置该变量，系统会依次查询配置变量 `core.pager`、环境变量 `PAGER` 以及内置的值。如果上述属性值不为空，那么系统会根据优先级规则进行相应的操作。

类似的情况也适用于 `GIT_EDITOR` 配置编辑器，交互模式下，当用户需要编辑某些文本时（例如添加一个提交注释信息），系统会启动相应的编辑器。注意，编辑器可以是一段用来生成相应输出结果的脚本，而非真正的编辑器程序。优先级靠后的环境变量根据系统环境的不同可以是 `EDITOR` 或者 `VISUAL`（在 `core.editor` 之后）。

## 环境变量对版本库地址的影响

Git 采用若干环境变量来决定它与当前版本库之间的接口。这些环境变量会对所有核心的 Git 命令产生影响。

例如环境变量 `GIT_DIR` 和 `GIT_WORK_TREE`，如果设置了上述变量属性值，即分别



指定了 `.git` 目录（包含版本库的管理区）的路径，以及对应的工作树（工作区）。命令行选项 `--git-dir` 也可以用来设置版本库的路径。工作树的路径可以通过选项 `--work-tree` 和配置变量 `core.worktree` 进行设置。

当然，如果版本库是非裸库，其中已经有一个工作区的话就另当别论。默认情况下，存放于 `.git` 文件夹下的版本库都是非裸库；用户还可以通过设置配置变量 `core.bare` 显式声明。

如果显式声明版本库的路径名，Git 将会从当前目录开始向上回溯目录树查找 `.git` 目录。如果找到了它，`.git` 所在的目录就会被当作工作树（项目的顶层目录），被找到的 `.git` 目录就是版本库的地址。用户可以通过 `GIT_CEILING_DIRECTORIES` 指定一组由冒号分隔的绝对路径列表（例如排除某个加载效率低下的网络路径），那么 Git 就不会遍历整个树目录了。一般来说，Git 会在文件系统边界停止上述遍历操作（除非将 `GIT_DISCOVERY_ACROSS_FILESYSTEM` 设置为 `true`）。

用户可以通过环境变量 `GIT_INDEX_FILE` 声明一个备用索引文件的路径，默认情况下会调用 `$GIT_DIR/index`。注意，索引不会出现在裸版本库中。该变量可以用来在不接触工作区的情况下创建或者修改一个提交的状态，即不接触文件系统。

变量 `GIT_OBJECT_DIRECTORY` 可以用来声明对象的存储路径；默认情况下使用的是 `$GIT_DIR/objects`。

此外，基于 Git 对象不可变的原生属性，以及它们是基于内容编址的事实，旧的对象可以被归档方便共享，只读目录可以存在于 `GIT_OBJECT_DIRECTORY` 之外。当然用户需要告知 Git 系统去哪里找到它们。或者换句话说，Git 版本库可以共享对象数据库（以及一些注意事项）。可以使用 `git clone --reference <repository> <URL>` 命令达到上述目的。这个问题的详情可以参考第 9 章。

用户可以设置环境变量 `GIT_ALTERNATE_OBJECT_DIRECTORIES`，指定可以用来搜索 git 对象的其他目录。这个变量可以指定一组由“:”分隔的路径列表，Git 得以在 `$GIT_DIR/objects/info/alternates` 文件和版本库自身数据库之外搜索相关对象。新增对象将不会被写入这些候选区域内。



#### 如何比较两个本地版本库？

现在假定用户希望比较两个本地版本库，但是由于某些原因，用户无法将其中的一个版本库配置为另外一个版本库的远程版本库来获取相关信息。解决这类问题的一种方案是使用只读存储。



对于其中的一个版本库，用户可以做如下配置：

```
GIT_ALTERNATE_OBJECT_DIRECTORIES=../repo/.git/
objects \ git diff \
$(GIT_DIR=../repo/.git git rev-parse --verify
HEAD) \ HEAD
```



对于另外一个版本库，用户需要获得对象的统一标识符，即 SHA-1 码。可以执行 `git rev-parse` 命令达到上述目的。为了转换一个引用，例如 `HEAD` 或者 `HEAD:README`，用户需要在另外一个版本库上执行相关操作，即通过环境变量 `GIT_DIR` 或者命令行选项 `--git-dir` 来完成（如示例中所示）。

## 环境变量对提交操作的影响

Git 提交对象的最终创建是由底层命令 `git commit-tree` 完成的。同时在命令行中，父提交的信息作为命令行参数传入，`git commit-tree` 命令的提交注释信息通过一个标准输入接收，作者和提交者的信息从下列环境变量中读取。

环境变量 `GIT_AUTHOR_NAME` 和 `GIT_COMMITTER_NAME` 分别是人性化的作者和提交者的信息字段。作者的电子邮件可以通过 `GIT_AUTHOR_EMAIL` 设置，提交者的电子邮件可以通过 `GIT_COMMITTER_EMAIL` 设置。如果配置变量 `core.email` 未设置，则退而求其次，使用通用的环境变量 `EMAIL`。`GIT_AUTHOR_DATE` 是一个时间戳，以 RFC 2822 电子邮件格式表示作者字段相关的日期（Fri, 08 May 2015 01:35:42 +0200），ISO 8601 是标准日期格式（2015-05-08T01:36:48+0200），Git 内部使用的时间格式是 UNIX 时间+hhmm 时区偏移（1431041884 +0200），或者其他 Git 支持的时间格式，其和 `GIT_COMMITTER_DATE` 类似。

在某些情况下，这些变量（部分）没有被设置，相关信息会从配置文件中读取，或者 Git 会尝试猜测这些信息。如果没有提供所需信息，Git 无法猜测它们，那么提交操作将不能成功执行。

## 8.2 重写修订历史

很多时候，在一个项目上进行研发工作时，用户可能希望对自己的提交历史记录进行修改。有可能是为了在提交变更到上游之前方便代码审核。另外一个原因可能是接受审核

者对下一版本的改进建议信息。或者是如第 2 章所述，用户希望保持项目历史记录整洁，以方便进行二分查找和回归分析。

Git 做得非常棒的一点就是支持历史记录的编辑和重写，并且提供了大量工具修改版本历史记录并保持其整洁性。



版本控制系统用户之间存在两个互相冲突的观点：一方认为历史记录一经产生，用户最好保持开发工作历史记录的原貌即可。另一方则认为在将历史记录发布之前，用户应该经常编辑历史记录，以提高它们的整洁性和可读性。

一个值得严重关注的问题是，即使我们在谈论重写历史记录，Git 内部的对象也是不可更改的（包括提交）。这意味着重写操作实际上只是创建了一个提交的副本，在修订 DAG 视图上新增了一个节点。相应的分支引用被切换到了刚创建的新节点上。原生的、准备重写的提交记录仍然存在于版本库中，通过引用日志仍然是可以访问的（或者通过 `ORIG_HEAD` 变量），至少在它们在垃圾回收过程中变成无法引用和不可达的对象之前是如此。当然，这也可能发生在引用日志过期之后。

## 8.2.1 编辑最后一次提交

历史记录重写最简单的情况是在分支上纠正最后一次提交对象中的错误（当前提交）。

有时用户在提交注释信息中发现了一个拼写错误，或者最后一次提交的修订版本不够完整。如果用户还没有推送（发布）自己的变更，那么是可以编辑最后一次提交的。用户可以使用 `git commit` 命令配合选项 `--amend` 一起使用达到上述目的。

提交被修改后的结果如第 3 章的图 3-6 所示。注意，修改最后一个提交和深入编辑提交的历史记录在功能上并没有差别。这两种情况下，用户都创建了一个新的提交，旧的提交只能通过引用日志才能访问。

这里索引再次显示了其强大威力（更明确一点是说提交对应的暂存区）。例如，如果用户只是希望简单地纠正提交的注释信息，并且不希望做任何变更，那么可以使用 `git commit --amend` 命令（注意该命令没有 `-a/--all` 选项）。该操作就像用户开始在新的提交上工作一样，至少假定用户没有对索引做任何变更。如果用户做了某些变更，那么可以使用 `git stash` 命令将提交暂时从暂存区拿走，修复最后一个提交的注释信息，然后提取暂存变更，使用 `git stash pop --index` 命令将它恢复到索引中。

另外一方面，如果用户发现自己忘记提交某些变更，用户可以编辑相关的文件，然后使用 `git commit --amend --all` 命令即可。如果变更之间存在交叉，用户可以试试用 `git add` 命令，或者使用它的交互式版本（详情可以参考第5章）创建用户希望添加的内容，最后执行 `git commit --amend` 命令即可。

## 8.2.2 交互式变基

有时用户可能会希望深入编辑提交的历史记录，或者将提交记录整合到一个存在逻辑关系的步骤中。用户在 Git 中可以解决这个问题的内置工具是 `git rebase --interactive` 命令。

这里，我们将假定用户使用的是第6章介绍和推荐的主题分支工作流，并且使用一个独立的主题分支研发某个功能特性；同时还假定用户正在处理的工作包含若干逻辑步骤，而非一个大型提交。

在实现一个新特性时，用户通常在工作之初并不能完美地完成所有工作。用户将会引入一系列自包含的小步骤（详情可以参考第12章），以便代码审核（代码审核）和二分查找（回归分析）更容易一些。通常用户只有在完成所有工作之后才会了解如何将它们更好地分隔。同时，希望用户在研发新特性时不犯任何错误也是不合理的。

在提交变更之前（推送变更到中心版本库，推送到自己的公共版本库并发送一个 `pull` 请求，或者采用第5章介绍的其他工作流），用户经常会希望将自己项目的当前分支保持最新的状态。通过在当前状态对自己的变更进行变基，并让它们及时更新。当维护人员（集成经理）接受用户提交的变更并将它们集成到主干分支时，可以让他们更容易地集成自己的变更。如前面章节所述，交互式变基能够让用户保持项目历史记录整洁性。

除了在发布变更之前清理变更记录之外，交互式变基这类工具还有其他用法。在研发一个比较复杂的功能特性时，首次提交的软件修订请求并不总是会被上游接受并添加到项目里面。一般来说，补丁中包含审核代码发现的问题或者变更记录的详细说明；又或者缺少某些东西（例如功能特性缺少说明文档或测试代码），某些提交需要修复，提交的一系列补丁（或者一个 `pull` 请求中的分支提交）应该被分割成更小的提交以便于代码审核。在这种情况下，用户还可以使用交互式变基来准备一个新版的提交请求，并且充分考虑代码审查后的修改建议。

### 提交的重排、移除和修复

如第7章所述，变基是将一系列提交的变更变基，然后将它们在一个新的基底（一个



新的提交)上重新应用。换句话说,变基是会移动的变更集,而非快照。Git 是在编辑器中打开重放变更的指令表对应的一系列操作启动交互式变基的。



用户可以通过配置变量 `sequence.editor` 设置变基指令文件的文本编辑器替换默认的编辑器,不过该配置变量可以被环境变量 `GIT_SEQUENCE_EDITOR` 重写覆盖。

和编辑注释信息的模板类似,指令表附带了注释信息,告知用户相关指令的具体用法(注意,如果用户使用的是旧版的 Git 程序,某些交互式变基命令可能会找不到):

```
pick 89579c9 first commit in a branch
pick d996b71 second commit in a branch
pick 6c89dee third commit in a branch

# Rebase 89579c9..6c89dee onto b8fffe1
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

如上述注释中所解释的那样,指令是按顺序执行的:从指令顶部开始创建第一个提交,并将新的基底作为其父提交;结束于底部拷贝被变基分支外部引用的提交的指令。这意味着修订是以时间先后顺序排序的,旧的提交排在前面。这和 `git log` 命令输出结果中先显示最近的提交的顺序相反(除非使用 `git log --reverse` 命令)。这一点很容易理解,变基会以它们被添加到分支上的顺序重新应用变更集,而查询日志操作显示的提交是从外部



引用开始到可达的提交进行排序的。

指令表的的每一行包含用空格分隔的 3 个元素。

首先是单词命令。默认情况下，交互式变基从 `pick` 命令开始。每个命令都有单个字母的快捷方式，如上述示例所示，用户可以使用单个字母代替该命令的完整形式（例如用户可以使用“`p`”代替“`pick`”）。

其次是一个可以唯一区分、用于执行命令的提交对象的 SHA-1 标识符。严格来说，它 是被变基提交对象的标识符，在变基开始之前就应该被获取了。该 SHA-1 标识符的简称用于选择相应的提交对象（例如在重排流水线时，它代表重排提交）。

最后是提交的描述信息（主题）。它是从提交对象注释信息的第一行内容提取的（具体来说，它是提交注释信息的第一个自然段，并且删除了最后的换行符。一个段落被定义成一组连续文本行，与其他段落区分开至少需要一个空行，即两个以上的尾行换行符）。这是提交注释信息的第一行最后是变更记录的简短描述的原因之一（详情可以参考第 12 章）。这个描述可以帮助用户决定如何处理该提交；Git 采用了它的 SHA-1 标识符并忽略了其余的内容。

在指令表中使用交互式变基对提交重排就和重排代码行一样简单。不过需要注意的是，如果变更集不是独立的，用户可能还需要解决冲突，即使在重排之前不存在合并冲突的情况下也是如此。在这种情况下，因为是 Git 进行操作的，用户需要修复冲突，并将冲突标记为已解决的（例如使用 `git add` 命令），然后执行 `git rebase --continue` 命令。Git 将会记录用户在交互式变基过程中的行为，因此用户不需要重复使用 `--interactive` 选项。

另外一个处理冲突的解决方案是通过 `git rebase --skip` 命令跳过冲突，而不是解决冲突，这也是上述命令的另外一种用法。默认情况下，变基会移除已经在上游中出现过的变更；用户也许会希望使用这命令来防止变基无法正确检测有问题的提交已经在将要移植的目标分支中出现了。换句话说，如果用户能够分辨一个冲突是一个空的变更集，那么可以直接跳过该提交对象。



用户可以使用 `git rebase --edit-todo` 命令，在任何时间继续执行由于某些原因而中断的指令表（例如指令表中包含一个错误，以及使用 `squash` 命令处理第一个提交等）。编辑好之后，用户可以继续执行变基操作。

为了移除变更，用户只需要简单地将指令表中相关行删除，或者将其注释掉，在最新版的 Git 程序中也可以使用 `drop` 命令。用户可以使用该命令丢弃失败的实验成果，或者通过删除上游中已经存在的变更集，使得变基操作更容易一些，不过这些变更集可能在形式上不尽相同。注意，移除指令表的过程中会终止变基操作。

为了修复一个提交，可以对指令表中 `pick` 命令之前的相关提交执行 `edit` 命令（或者使用代码 e）。这会让变基操作停在这个需要编辑的提交上，即重新应用变更的步骤，情况和解决冲突类似。更精确地说，交互式变基遇到有问题的提交，因此它是 HEAD 提交，终止变基过程，将控制权让渡给了用户。接下来用户就可以修复该提交了。如前所述，用户可以像用 `git commit --amend` 命令编辑当前提交那样对上述提交进行修复。修复好上述提交之后，如 Git 指令帮助中所述，执行 `git rebase --continue` 命令就可以继续变基过程了。

一个适当的 git 感知命令行提示工具，诸如 Git 的 `contrib` 工具，它会告知用户是否处于变基过程中（详情可以参考第 10 章）。如果用户没有使用这样一个命令行提示工具，可以使用 `git status` 命令查看系统当前状态：



```
$ git status
rebase in progress; onto b3cebef
You are currently rebasing branch 'subsys' on
'b3cebef'.
(fix conflicts and then run "git rebase
--continue")
(use "git rebase --skip" to skip this patch)
(use "git rebase --abort" to check out the
original branch)
如你所见，用户总是可以使用 git rebase --abort
回到变基之前的状态。
```

如果用户只想修改提交的说明信息（例如修复拼写错误或者添加额外的信息），用户可以跳过执行 `git commit --amend` 命令的步骤，执行 `git rebase --continue` 命令之后，使用 `reword` 命令代替 `edit` 命令。Git 将会用编辑器自动打开提交的注释信息文件。保存相关变更然后退出编辑器之后，系统会自动提交这些变更，修改相关的提交对象并继

续执行变基操作。

## 压缩提交

有时用户可能需要将两个或者多个提交整合到一起。也许用户觉得不应该将它们拆分，放在一起会更好一些。

通过交互式变基，如果有必要的话，用户可以对提交对象进行重排，使得提交之间挨得更近，然后对于第一个希望被连接到一起的提交取消执行 `pick` 命令（或者对它执行 `edit` 命令）。对于剩下的提交，使用 `squash` 或者 `fixup` 命令。Git 将会整合这些变更，然后为上述所有提交创建一个提交。建议被折叠的提交的注释信息选用其中第一个提交的注释信息，并在调用 `squash` 命令时进行添加；执行 `fixup` 命令时注释信息会被自动忽略。这意味着 `squash` 命令擅长压缩提交，`fixup` 命令擅长添加修复注释信息。如果提交之间的作者不同，被折叠的提交将会选用第一个提交的作者信息。提交者将会是当前用户、当前执行变基的人。

假定用户发现自己忘记为提交添加部分变更信息，例如忘记添加测试程序（或者只进行了负面测试）或者说明文档。提交已经存在于历史记录中，因此用户不能只通过编辑对它进行添加。用户可以使用交互式变基或者补丁管理接口修复它，但是常用的更高效的做法是创建一个包含被遗忘变更的提交，然后将它们压缩。类似地，当用户发现自己刚才创建的提交中有一个 bug，可以创建一个包含 bug 修复的 `fixup` 提交，然后将它们压缩。

如果用户采用了这种技术，有时可能必须在发现需要附加新的变更或者修复一个 bug 和创建一个合适的提交之间疲于奔命，然后还需要花时间进行变基操作。如何将上述提交标记成 `squash` 或者 `fixup` 呢？如果用户在提交注释的开头分别使用魔法字符串 `squash! ...` 或者 `fixup! ...`，并且在将要被压缩的提交的描述信息之前，用户可以要求 Git 自动压缩它们。这还会自动修改 `rebase -i` 命令的计划列表。用户可以在一个独立的基础对象上使用 `--autosquash` 选项，或者通过配置变量 `rebase.autoSquash` 默认启用该功能。为了创建相应的魔法提交注释信息，用户可以使用 `git commit --squash/--fixup` 命令（将被压缩/被修复的提交需要作为该命令的参数进行传递）。

## 拆分提交

有时用户也许希望将一个或者两个提交拆分成两个以上的多个部分。也许你已经注意到提交太大了，尝试完成的工作太多了，最好将它们一分为二。又或者决定将变更集的某一部分从一个提交移动到另外一个上，将它们提取到另外一个独立的提交上是完成该工作的第一步。



Git 并没有内置处理这类问题一步到位的命令。然而，巧妙地使用交互式变基完全能够满足拆分提交的需要。

如前所述，为了拆分一个给定提交，首先需要将它对应的动作标记为 `edit`，Git 会在特定的提交停下，将控制权让渡给用户。对于分割提交的情况，当用户使用 `git rebase --continue` 命令将控制权让渡给 Git 时，用户预期的目标是用两个提交替换原来的一个提交。

拆分提交的问题堪比第 3 章提及的变更杂糅在一起的问题（交互式提交章节）。区别在于提交已经创建并且从被变基的分支中拷贝了出来。用户可以非常简单地使用 `git reset HEAD^` 命令对它进行修复；如第 4 章所述，这个命令将在移动 `HEAD` 指针的同时，保留提交被分割之前工作区中的状态（变更杂糅），以及这个修订之前提交在暂存区的状态。然后用户就可以交互式地将希望添加到第一个提交中的变更记录添加到索引中，在暂存区中整合中间步骤。接下来，检查索引中是否有用户想要的内容，并根据从中提取的内容使用不带 `-a` / `--all` 选项的命令 `git commit` 创建一个新提交。重复最后两个步骤经常是必要的。

另外，除了交互式添加变更，用户可以交互式地移除变更为拆分提交创建中间步骤。如第 4 章所示，可以使用交互式重置达到此目的。

对于一系列提交中的最后一个提交（如果用户是将提交一分为二，那么就是指第 2 个提交），用户既可以添加任意内容到索引，保持工作区的整洁性，根据索引中的内容创建一个提交，也可以根据工作区的状态创建一个提交（`git commit -a` 命令）。如果用户希望保留或者启动，将原生提交的注释信息进行拆分，那么可以在创建提交时使用 `--reuse-message=<commit>` 或者 `--reedit-message=<commit>` 选项。

最简单的拆分一个具名提交的方式是使用引用日志——它只是重置之前的 `HEAD@{n}` 实体：在 `git reflog` 的输出中移动 `HEAD^` 的位置。

除了在暂存区（索引）从被拆分提交的父提交开始创建提交，交互式添加变更之外，用户还可以从最终状态启动（即将被拆分的提交），并在第 2 步中移除变更，例如使用 `git reset --patch HEAD^` 命令（交互式移除），坦白地说，用户可以使用第 4 章介绍的意义技术组合。另外，图形化的提交工具（如 `git gui`）非常有用（图形化提交工具的详情可以参考第 11 章）。

如果用户对于在索引中创建的中间步骤修订版本的一致性没有绝对把握（编译，通过测试用例等），用户可以使用 `git stash save --keep-index` 命令暂存还未准备好的变更，将工作区和索引中的状态进一步整合。用户可以测试这些变更（例如使用测试用例），如果必须修复相关变更，那么可以修改暂存区中的内容。或者用户可以根据索引创建提交，



然后在创建每个提交之后单纯地使用 `git stash` 命令保存工作区的状态。如果必须修复相关提交，用户可以测试和编辑现存的中间步骤提交。在上述两种情况下，用户在拆分一个新的提交之前，都需要使用 `git stash pop` 命令恢复变更。

## 测试已变基提交

一个好的软件开发实践是在提交每个变更之前对它们进行测试。不过并非总是如此。假定用户忘记测试某些提交或者直接跳过了它，因为这些变更看起来变化不大，并且用户又急于将软件上线。交互式变基支持用户执行测试（更精确地说是任何命令），在执行变基过程中将 `exec (x)` 动作添加到用户希望测试的提交后面。

`exec` 命令会在 `shell` 中启动相关命令（该行中其余的指令），即环境变量 `SHELL` 中声明的应用程序，如果 `SHELL` 未设置，那么将会调用默认的 `shell` 程序。这意味用户可以使用 `shell` 特性（对于 `POSIX shell`，它会使用 `cd` 改变目录位置，用 `>` 对命令输出进行重定向，使用 `;` 和 `&&` 拼接多个命令字符串等）。非常重要的一点是，将要执行的命令是在工作区根目录下执行的，而非当前目录（在启动交互式变基时用户所在的子目录）。如果用户严格遵循未通过测试的变更不予发布的原则，可能会在实际的重写提交、在新的变更上进行变基等方面无法通过测试，即使原生提交能够通过测试也是如此。不过用户可以使用 `--exec` 选项在交互式变基过程中测试每个提交，例如下列代码：

```
$ git rebase --interactive --exec "make test"
```

这会修改指令表的启动过程，在每个实体之后插入 `exec make test` 命令：

```
pick 89579c9 first commit in a branch
exec make test
pick d996b71 second commit in a branch
exec make test
pick 6c89dee third commit in a branch
exec make test
```

## 8.2.3 外部工具——补丁管理接口

用户也许会希望在历史提交中发现 `bug` 时马上将它修复，而不是推迟到该分支被变基之后再处理。后者通常只会在分支被提交审核（准备发布）之前发生。这可能在意识到需要变基过去的提交时会经历不短的时间跨度。

Git 本身并没有提供直接修复 `bug` 的简易方式，也没有内置的工具可供选择。不过用户可以使用第三方的外部工具，在 Git 的基础上实现补丁管理接口。这类工具包括 `Stacked Git`

(StGit)和 Git Quilt (Guilt)。

这些工具提供了和 Quilt 类似的功能(用堆栈管理补丁)。通过它们,用户可以在类 Quilt 堆栈中保存一组处于研发状态的浮动补丁。还可以通过 Git 提交的形式接收变更记录。用户可以在补丁和提交之间互相转换,移动和编辑补丁,移动和编辑提交(将提交和其子提交转换成补丁,或者再转换回来),压缩补丁等。

不过需要注意的是,用户需要安装额外的工具,还要花费时间学习上述工具的使用(即使它们能够将你的工作变得更简单);Git 和这些工具之间的边界产生的复杂性也是用户需要预计到的。目前交互式变基和自动压缩功能已经足够强大,Git 另外一个层面的需求也相应减少了。

## 8.2.4 使用 git filter-branch 进行脚本化重写

在某些情况下,用户可能还需要使用比交互式变基更强大的工具来重写修订历史并保持其整洁性。用户可能还需要能够重写整个修订历史的工具,或者非交互式的重写工具,采用某些特定的算法执行重写操作。能够应付这种情况的是 git filter-branch 命令。

这个命令的调用方式和交互式变基完全不同。首先,用户需要给它指定一个或者一组需要被重写的分支,例如--all 选项是重写整个分支。严格意义上来说,用户是把 rev-list 给它当作参数的,即一系列的正负引用(详情可以参考第 2 章)。命令只会重写命令行中声明的正引用。这说明正引用是修订区间的上界,需要具备被重写的能力,即成为分支名。负修订用来约束重写过程的修订范围。当然,用户也可以在命令行中声明一个特定路径来指定修订区间。

该命令会通过每个将被重写的修订应用用户自定义过滤器来实现对 Git 修订历史的重写。这是另外一个差异:变基通过重新应用变更集来达成目标,分支过滤通过快照达到目标。这种做法的后果之一是,对于分支过滤来说,合并只是一种提交对象,而变基会跳过合并,如果用户在交互式变基过程中遇到问题,可以使用--preserve-merges 选项处理。

当然,对于分支过滤,用户是使用脚本达到重写的目的(有时也称过滤器),而不是交互式重写——编辑指令表,在变基过程中,对提交手动执行编辑、改写、压缩、拆分和测试等命令。这意味着分支过滤操作的执行速度不会受到人机交互的影响,只和 I/O 有关。建议用户使用 -d <directory> 为重写指定一个非硬盘式的临时目录(如果过滤器需要的话)。



因为执行 `git filter-branch` 命令通常涉及大量的重写操作，它会保存原生引用，即在 `refs/original/` 命名空间下指向预重写的历史记录（用户可以通过 `--original <namespace>` 将它覆盖）。

如果 `refs/original/` 中已经存在起始引用，或者临时目录中已经包含若干内容，除非用户强制执行，否则系统会拒绝启动命令。

## 无过滤器的分支过滤

如果不指定过滤器，提交对象将以没有任何变更的形式被再次提交。这类操作一般是无效的，但是它可以用来在未来某个时间修复 Git 中的 bug。需要注意的是该命令对移植（是 `.git/info/grafts` 文件）和置换集（命名空间 `refs/replace/` 下的引用）的影响，用户可以通过命令行选项禁用后者。移植（grafts）和置换是在不重写任何历史修订的情况下影响历史记录两种方法。本章后续内容将会详细介绍它们。

这意味着执行不带任何过滤器的 `git filter-branch` 命令，可以用来对移植或者置换操作重写给定提交记录时施加永久性的影响。通过这种方法，用户可以使用如下技术：在特定提交对象上使用 `git replace` 命令来修改历史记录的视图，以便确保其正确性（根据用户的需要进行修改），然后使用分支过滤对其进行永久性的修改。

此外，在重写提交对象时，`git filter-branch` 命令会受到若干相关的配置变量当前值的影响。自原有提交被创建到重写这些提交时，这些变量的值可能已经发生了改变。这一特性可能会有助于修复历史记录，例如当用户使用了非标准的编码格式编写提交的注释信息时（非 `utf-8` 格式），却忘记了设置 `i18n.commitEncoding` 变量的值。通过无过滤器的历史重写操作，将 `i18n.commitEncoding` 设置为正确的值之后，将会在提交对象的 `encoding` 首部添加上述变量的值。

## 分支过滤的可用过滤器类型

有大量的不同种类的过滤器供用户声明如何重写历史记录。用户可以一次性声明多种过滤器，它们会按照排列的顺序依次被应用。注意不同的过滤器有不同的执行性能考虑。

一旦对提交启动了重写操作，命令行参数都会被 `shell` 上下文解析并执行。当前的预重写提交（即将要被重写的提交对象）的 `SHA-1` 标识符相关信息将会通过环境变量



GIT\_COMMIT 传递到过滤器。另外，有一个 shell 映射函数可以将原生提交的 SHA-1 码转换成命令行参数，其输出结果是被重写后的提交还是原生提交的 SHA-1 码，取决于提交被重写时系统是否调用了 shell 函数。

当然，环境变量 GIT\_AUTHOR\_NAME、GIT\_AUTHOR\_EMAIL、GIT\_AUTHOR\_DATE、GIT\_COMMITTER\_NAME、GIT\_COMMITTER\_EMAIL 和 GIT\_COMMITTER\_DATE 的值都是从当前提交对象中获取的，使得写入内容到过滤器更方便一些，并且会对替代提交的作者和贡献者产生影响。如果过滤器执行成功，分支过滤会使用 git commit-tree 命令创建一个替代提交；如果该命令返回一个非 0 的退出状态码，那么整个重写过程将会终止。

过滤器脚本的编写和普通脚本大致类似，通常它和专门用于人机交互的高层命令相比，能够更方便地使用底层命令。具体来说，分支过滤命令自身就是底层命令，其中不包含任何人性化的功能（和下面的 gitignore 文件类似）。如果用户喜欢的话，可以使用过滤器程序代替 shell 脚本。

git filter-branch 命令支持的过滤器类型有以下几种：

- `--env-filter`：它可以用来修改被重写提交的执行环境。用户可以用它来修改作者或者提交者信息、开发人员的名字、电子邮件、时间等。注意，那些变量需要重新导出。
- `--tree-filter`：它可以用来重写提交对象的内容，即提交对象引用的树对象。在工作目录是项目的根目录并且当前提交处于签出状态下，该命令会被 shell 解析执行。当该命令执行完毕，工作区的内容也会发生相应变化，在不考虑任何忽略规则的情况下（例如来自 .gitignore 文件），新增的文件将会自动添加，消失的文件会被自动移除。
- `--index-filter`：它可以用来重写将要被重写的提交对象相关的索引和暂存区。它通常是树过滤器，不过速度更快，因为它不需要将文件签出到工作区（文件系统）。
- `--parent-filter`：它可以用来重写提交的父提交列表。它会接收到一个以命令行参数形式的父提交字符串并以一个标准输入传递给 git commit-tree 命令（-p <parent full SHA-1>），然后以标准输出格式返回一个新的父提交字符串。
- `--msg-filter`：它可以用来重写提交的注释信息。它会以标准输入的形式接受原生提交的注释信息，以标准输出的形式返回一个新的提交注释信息。
- `--commit-filter`：它可以用来替代 git commit-tree 命令声明希望调用的



命令。这意味着获得<tree> [(-p <parent>)...]参数到过滤命令, 以及以标准输入的形式获得日志信息。

用户可以使用它实现一些很有用的功能: `skip_commit "$@"`忽略当前提交(但是不忽略它的变更!), `git_commit_non_empty_tree "$@"`可以忽略没有发生变更的提交。



"\$@"代表命令中以第1个位置开始的位置参数。当表达式遇到双引号时, 每个参数会转换成独立的字符。这是一个标准的 POSIX shell 特性, 可以用来确保向下传递参数时参数不发生变更。

- `--tag-name-filter`: 它可以用来重写标签名称。原生的标签名称是通过标准输入的形式传递的, 相关命令会以标准输出的形式返回新的标签名称。原生的标签并未被删除, 不过可以被重写; 用户可以方便地使用`--tag-name-filter cat`选项更新标签内容(剥离签名)。

注意剥离签名的操作, 因为根据定义, 它不能被保留。标签的重命名会相应地重写指向发生变更的目标。目前其还不支持修改便签、时间戳、标签信息和标签重签名。

- `--subdirectory-filter <directory>`: 它可以用来只保留给定目录下的项目历史记录, 并可以将该目录设置为项目根目录; 可以用来将项目的子目录转换成一个子项目, 详情可以参考第9章。

注意, 如果用户使用 `git log` / `git rev-list` 是通过选项参数限制被重写修订的数量时(例如`--all`选项会重写所有分支), 用户必须使用`--`符号将特定的过滤器和其他分支过滤选项分隔开。

### git filter——branch 命令应用示例

假定用户因为失误提交了一个错误的文件到版本库中, 现在希望将该文件从项目历史记录中移除。也许这是一个包含密码等敏感信息的特定网站配置文件。可能是用户执行 `git add` 命令时没有配置好忽略文件属性(也许它是一个大的二进制文件)。希望事后证明用户并没有发布文件的权限, 最好将它们删除, 避免侵犯版权。

现在用户需要将它从项目中移除。使用 `git rm --cached` 命令移除该提交之后, 不

受它影响的仅限于未来的提交。用户还可以通过修改该提交，将该文件从最新版的提交中移除（详情可以参考本章前面的章节）。为了将该文件从实际的历史记录中剔除（假定该文件名是 `passwords.txt`），用户可以使用 `git filter-branch` 命令搭配 `--tree-filter` 选项一起使用来达到此目的：

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite fdfb73095fc0d594ff8d7f507f5fc3ab36859e3d (32/32)
Ref 'refs/heads/master' was rewritten
```

当然，还有比使用树过滤器更高效的解决方案，即引入文件输出机制，用户可以使用 `git rm --cached` 命令配合索引过滤器将上述文件从索引中删除。务必确保过滤器能够成功执行，不会因为遇到异常而中途退出，即使没有文件可以删除也是如此；同时也不需要返回输出信息：

```
$ git filter-branch --index-filter \
    'git rm -f --cached -q --ignore-unmatch passwords.txt' HEAD
```

用户也可以采用后续章节介绍的第三方工具 **BFG Repo-Cleaner**。

用户可以使用分支过滤移除历史记录中特定类型的提交，例如某个特定作者创建的提交（例如某个没有履行著作权义务的作者，例如贡献者协议）。注意，使用分支过滤和交互式变基在移除提交方面有一个非常重要的区别。分支过滤会移除修订 DAG 视图上的节点，但是不会移除相关的变更记录——因为它们不再是两个快照之间的中间步骤了，并且变更移动到了子提交。换句话说，交互式变基会移除提交和相应的变更。这意味着所有子提交都被修改了，它们的快照不再包含被移除的变更记录。

为了移除一个提交，用户可以在提交过滤器中使用 `skip_commit` 的 shell 函数：

```
$ git filter-branch --commit-filter '
if [ "$GIT_AUTHOR_NAME" = "Bad Contributor" ];
then
    skip_commit "$@"
else
    git commit-tree "$@"
fi' HEAD
```

用户可以使用分支过滤永久性地整合两个版本库，串联历史记录，将历史记录一分为二。可以使用父过滤器直接达到上述目的。例如整合两个版本库，将其中一个版本库历史记录中包含根节点 `<root-id>` 的提交和包含移植 `<graft-id>` 的提交串联起来（来自另外

一个版本库)，并将包含移植 id 的提交作为父节点，相关代码如下：

```
$ git filter-branch --parent-filter \
'test "$GIT_COMMIT" = <root-id> && echo "-p <graft-id>" || cat' HEAD
```

用户可以通过一种非常简单的方式将给定提交的历史记录一分为二，即将父节点通过 `echo ""` 命令设置为空集即可。

如果用户知道自己的提交记录中值包含一个根提交（唯一的一个没有父提交的节点），可以使用如下命令简化合并历史的步骤：

```
$ git filter-branch --parent-filter 'sed "s/^\$/-p <graft id>/' HEAD
```

不过，使用移植或者置换机制更简单一些。检查串联/拆分的历史记录是否能够正确渲染，然后通过不带过滤器的分支过滤对它们进行永久替换，分支过滤时需要指定修订区间的起点，至少要从重写的串联/根提交开始计算。当然，如果用户可以通过编程方式告知系统拆分（串联）哪个修订的话，`--parent-filter` 选项就能派上用场了。这种技术的简化版就是前面串联单一根节点示例中所采用的方案。

另外一种常见的情况是修复提交对象中错误的名字或者电子邮件地址。也许用户在工作之初忘记执行 `git config` 命令配置自己的用户名和电子邮件地址了，Git 系统会猜想这些信息可能不正确（如果系统不能猜测它们，会在用户创建提交之前要求用户输入这些信息），`.mailmap` 文件中的信息也不完备。

又或许用户希望公开以前是闭源程序的源代码，将它们内部的公司电子邮件地址改成个人的电子邮件地址。不管怎么说，用户可以通过分支过滤对历史记录中的所有电子邮件地址进行修改。用户只要确保正在修改的提交对象的正确性即可。可以通过 `--env-filter` 选项达到上述目的（当然 `--commit-filter` 也行，只是需要执行 `git commit-tree "$@"` 命令，并且没有导出行）：

```
$ git filter-branch --env-filter '
if test "$GIT_AUTHOR_EMAIL" = "joe@localhost"; then
    GIT_AUTHOR_NAME="Joe Hacker"
    GIT_AUTHOR_EMAIL=joe@company.com
    export GIT_AUTHOR_NAME GIT_AUTHOR_EMAIL
fi' HEAD
```

该示例展示了一个简化版的解决方案，用户肯定也会希望能够修改提交者的信息，相关的代码几乎是一致的。



如果用户开源了一个项目，那么用户也许需要添加签名信息：原生程序的数字认证信息（详情可以参考第 12 章）：

```
$ git filter-branch --msg-filter \
'cat && echo && echo "Signed-off-by: Joe Hacker <joe@company.com>"' \
HEAD
```

假如用户发现某个子目录的名称有拼写错误，例如 `include/` 被写成了 `inlude/`。如果想对它进行重命名，那么用户可以使用 `--tree-filter` 选项达到和 `mv -f inlude include` 命令类似的目的。不过，巧妙地使用 `--index-filter` 选项效率更高：

```
$ git filter-branch --index-filter '
git ls-files --stage |
sed -e "s!\\(\\t\\)*\\!inlude/!\\linclude/!" |
GIT_INDEX_FILE=$GIT_INDEX_FILE.new \
git update-index --index-info &&
mv "$GIT_INDEX_FILE.new" "$GIT_INDEX_FILE"
' HEAD
```

上述代码的含义如下：我们使用 `git ls-files--stage` 命令的实际输出结果来匹配 `git update-index --index-info` 命令的输入格式（后者是高层命令 `git add` 的一个底层命令）。为了替换文本并修复路径名中的拼写错误，采用了 `sed` 程序（流编辑器）。这里需要注意，编写正则表达式时某些文件名需要用到引号。一个临时索引文件用来执行一个原子操作。

一般来说，某些大一些的项目中的部分内容在其启动之初就有意和项目自身保持独立。我们希望提取出这部分的历史记录，让孩子目录成为一个新的根目录。为了以这种方式重写历史记录并丢弃其他不相关的历史记录，用户可以执行如下命令：

```
$ git filter-branch --subdirectory-filter lib/foo -- --all
```

当然，也许更好的解决方案是使用一个特定的第三方工具，例如 `git subtree`。这种工具（以及同类产品）将在第 9 章详细介绍。

## 8.2.5 用于重写大型项目历史记录的外部工具

`git filter-branch` 命令不是重写大型项目历史记录唯一的解决方案。还有不少其他更易用的工具供用户选择，有可能是因为它们包含大量的预定义清理操作，也有可能是因为它们提供了某个层面的交互式脚本重写能力（读取-解析-打印-循环（REPL），和某些



解释性编程语言的交互式 shell 程序类似)。

## 使用 BFG Repo Cleaner 从历史记录中移除文件

BFG Repo Cleaner(简称 BFG)是一款简单、高效,特别适合替代 `git filter-branch` 命令的工具,它可以帮助用户清理 Git 版本库历史记录中的无效数据:移除文件和目录,替换文件中的文本(例如密码占位符)。它在相关应用领域的执行效率要比分支过滤高,因为它会假设用户不会在乎目录层级中无效文件的所在位置,只是希望将它移除即可。此外,它还支持多核的并行处理,并且不需要提取版本库和为每个提交提供 `exec` 的 shell 执行过滤脚本。BFG 是由 Scala 语言编写的,并且使用 JGit 作为它的 Git 实现。

大部分情况下, BFG 的使用更简单,因为它专门为移除文件和修复文件提供了一组命令行参数,例如 `--delete-files` 和 `--replace-text`, 以及各种各样的查询语言。当然它缺乏分支过滤的灵活性(通常不是必需的)。

用户需要注意的一个问题是, BFG 假定用户修复的内容是针对当前提交的。

## 使用 reposurgeon 编辑版本库历史记录

reposurgeon 原本是用来清理版本库迁移过程中产生的冗余文件(从一种版本控制系统迁移到另外一种)。它依赖于 `git fast-import` 格式,并能够解析、修改和执行命令流。得益于版本控制的无限潜力,目前这种格式已经成了版本库系统中比较常见的导入导出格式。

它可以用于重写历史记录,其中包括编辑过去的提交和元数据、移除提交、压缩(合并)和分割提交。它还可以用于从历史记录中移除文件和目录,分割和串联历史记录等。

reposurgeon 和使用 `git filter-branch` 命令相比主要的优点在于,它有两种执行模式:既可以是交互式解析器,可以用作历史记录的调试器/编辑器,也支持命令历史记忆和 `tab` 键自动完成功能。批量模式可以将命令当作参数批量执行。这个特性使得用户能够交互式查看历史记录和测试变更,然后为所有修订批量执行相关操作。

它的缺点是必须进行程序安装,并且需要额外花时间学习该工具的具体使用。

### 8.2.6 重写已发布历史的风险

这里有一个非常重要的原则,那就是用户永远不要重写已发布的历史记录(至少不存在一个非常充分的理由),特别是对于那些已经发布到公共版本库上的提交或者因为某些原因被公开的提交。用户可以编辑的内容是修订视图上私有的部分。

这个原则背后的原因是,如果下游开发者根据被重写的历史记录进一步开展工作,可

能会给他们制造麻烦。

这意味着安全地重写和重新编译公共分支的时机是项目状态稳定，文档持续更新。例如将它们作为一种显示工作进度的方式（例如计划更新式的分支）。另外一个安全地重写公共分支的情况是在项目生命周期的某个特定状态，即软件发布新版本之后，以及添加文档说明时。

## 重写上游的后果

现在将通过一个简单的重写已发布历史记录示例（例如变基），向读者展示该操作产生的不良影响。现在假定用户需要处理两个公共分支：**master** 和 **subsys**。后者是基于前者构建的（提取）。还假定某个下游开发者根据 **subsys** 分支为自己创建了一个主题分支，不过并没有将它发布；该分支只存在于其自己的本地版本库。这种情况如图 8-1 所示（C1~C12 表示只存在于下游开发者的本地版本库中的本地分支）：

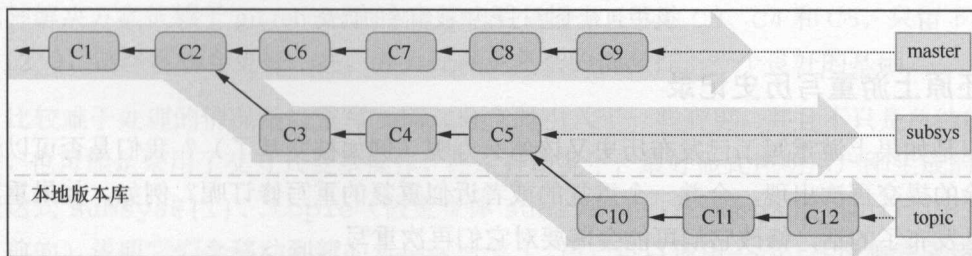


图 8-1 下游开发者在主题分支上添加新的变更，重写已发布历史提交之前本地版本库的状态

如图 8-2 所示，然后上游根据 **master** 分支当前修订（最顶层）重写了 **subsys** 分支，该操作就是第 7 章介绍过的变基操作。在重写过程中，其中某个提交被删除了：也许是因为 **master** 分支中出现了相同的变更而被忽略了，也可能是因为该提交通过交互式变基被压缩到之前的提交而被删除了（该操作会在后面交互式变基章节详细介绍）。

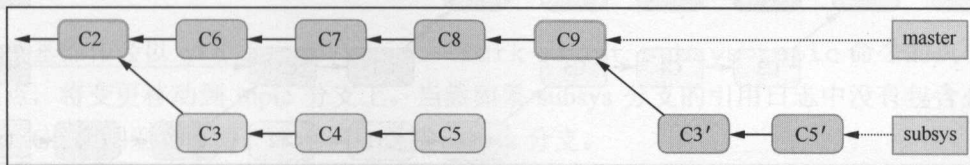


图 8-2 上游公共版本库的状态。用户可以看到着重强调的变基分支之后的原有基底，新的基底，以及被重写的提交（变基之后）

注意，在默认配置下，Git 会拒绝重写历史（将会拒绝执行非快进式推送）。用户需要进行强制推送。

问题在于合并操作是基于修订的预重写版本，例如本示例中的主题分支：

如果下游开发者和上游维护人员都没有发现已发布的历史被重写了，合并了来自主题分支的变更，例如基于 `subsys` 分支的提交，那么这时合并操作会添加重复提交到版本库中。如图 8-3 所示，经过上述合并操作之后（这里用 M13 表示），我们从主题分支得到了预重写提交 C3、C4 和 C5，以及后续重写提交 C3' 和 C5'。注意，在重写过程中被移除的提交 C4 又回来了，这可能是一个安全 bug！

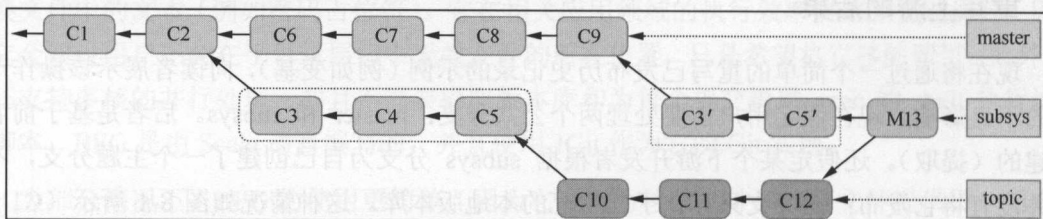


图 8-3 将预重写的版本合并到后续重写分支后的情况。注意，合并会添加预重写版本的修订，其中包括变基过程中被删除的提交

### 还原上游重写历史记录

但是如果上游重写了已发布历史又该怎么办呢（例如被变基了）？我们是否可以避免被移除的提交再次出现，合并一个重复的或者近似重复的重写修订呢？例如，如果重写修订意见发布了的话，修改它们可能会需要对它们再次重写。

解决方案是在上游将用户的工作变基，以便它能够匹配新的版本，将它从上游预重写修订移动到后续重写修订上。

如图 8-4 所示，在本示例的情况下，将会在新版本（后续重写）的 `subsys` 分支上对主题分支进行变基。

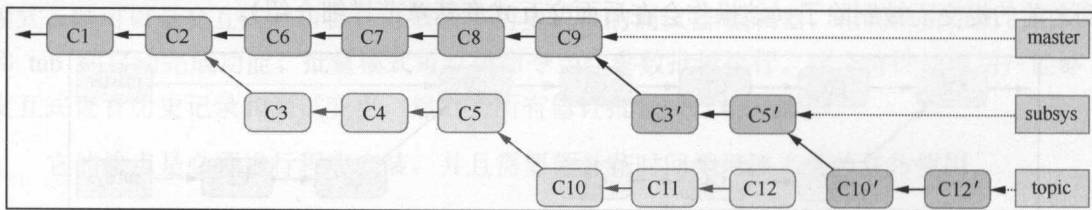


图 8-4 一个主题分支在下游变基后，可以在上游重写记录中恢复相关记录



也许用户还没有 `subsys` 分支的本地副本；在这种情况下，使用相应的远程跟踪分支替代 `subsys` 分支。  
例如 `origin/subsys`。



用户要根据主题分支公开与否灵活应对，这意味着现在用户可能要违背不改变下游公共历史记录的承诺。

还原一个上游重写记录可能会导致下游一石激起千层浪、发生连锁反应，执行一系列的变基操作（取决于版本库）。

分支 `subsys` 的变基属于非常简单的情况，变更前后是一致的（这意味着 `C4` 消失了，因为 `C6~C9` 中的某个提交已经包含了它），然后用户可以简单地对上游主题分支变基，即 `subsys`。相关代码如下：

```
$ git rebase subsys topic
```

如果用户目前正处于主题分支上（假定当前分支是 `topic`），那么上述命令中的 `topic` 并不是必需的。这就是变基涉及的对象：旧版本的 `subsys` 和用户在 `topic` 分支上的提交。不过这种解决方案依赖于 `git rebase` 命令会忽略重复提交（移除 `C3`、`C4` 和 `C5`，只留下 `C10` 和 `C12`）。这样可能会更好一些，并为处理更复杂的情况打下一个良好的基础。

比较难于处理的情况是当重写 `subsys` 分支时引入了一些变更，并且不只是单纯的变基操作，也可能是采用了交互式变基操作。在这种情况下，最好显式声明用户刚生成的变更，即表达式 `subsys@{1}..topic`（假定实体 `subsys@{1}` 在 `subsys` 的引用日志中是在重写之前的），说明它们会移动到新的 `subsys` 分支上。用户可以使用 `--onto` 选项完成该操作：

```
$ git rebase --onto subsys subsys@{1} topic
```

用户还可以让 `Git` 使用引用日志找到一个更佳的共同祖先，辅以选项 `--fork-point`，更好地进行变基操作：

```
$ git rebase --fork-point subsys topic
```

变基操作会以 `git merge-base --fork-point subsys topic` 命令的执行结果为起点，将变更移动到 `topic` 分支上。当然如果 `subsys` 分支的引用日志中没有包含必要的信息，`Git` 会回退到上游，本示例中是指 `subsys` 分支。

#### 注意：



用户可以像前文所述，使用交互式变基代替普通的变基操作，这样可以更好地控制工作成本（例如丢掉已经存在的提交，而不是由变基操作机械地进行检测）。



## 8.3 历史记录的非重写式编辑

如果用户希望修复已发布历史记录中的部分错误，又该如何做呢？如重写已发布历史记录章节所述，修改公共历史记录中的部分内容会对下游开发者产生不良影响（实际上是创建一个变更拷贝和替换引用）。用户最好不要碰修订视图中的这部分内容。

这个问题的解决方案非常少。常用的办法是添加一个新的、包含相应变更的 `fixup` 提交（例如修复文档中的拼写错误）。如果用户打算移除变更记录，至少找出它们必须被移除的若干证据才行，用户可以创建一个提交来还原上述变更。

如果用户修复或者还原了一个提交，那么它应该包含和该 `bug` 有关的详细说明信息，以及指明哪个提交被修复（还原）了。即使当提交是公共的以至于用户无法为被修复的提交添加注释信息也是如此。Git 提供了一种为现存提交记录添加额外信息的笔记机制，它类似于出版附录修订和勘误表。不过用户需要谨记，默认情况下笔记是不会被公开发布的，不过它们的发布也是非常容易的（用户只需要记住这一点即可）。

### 8.3.1 还原提交

如果用户希望备份一个现存的提交记录，撤销其中的变更，那么可以使用 `git revert` 命令。如第7章所述（例如图7-4），还原操作创建了一个变更记录颠倒的提交。例如原生的提交添加了一行内容，反向提交中会删除它；原生提交中删除了一行内容，反向提交中会添加它。

#### 注意：



不同版本控制系统对名为 `revert` 的对应操作是不一样的。具体来说，常见的含义是将文件的变更重置到最近一次提交的版本，并会丢弃未提交的变更。在 Git 中，它的含义和 `git reset -- <file>` 命令的作用类似。

最好使用一个例子来讲解该概念。以 `multiple` 分支上最新的一个提交对象为例，先查看其中变更记录的摘要：

```
$ git show --stat multiple
commit bb71a804f9686c4bada861b3fcd3cfb5600d2a47
Author: Alice Developer <alice@company.com>
```

```
Date:    Sun Jun 1 03:02:09 2014 +0200
Support optional <count> parameter
```

```
src/rand.c | 26 ++++++-----
1 file changed, 21 insertions(+), 5 deletions(-)
```

还原这个提交之后会创建一个新的修订版本（需要工作目录保持整洁）。这个修订会撤销被还原提交产生的变更记录：

```
$ git revert bb71a80
[master 76d9e25] Revert "Support optional <count> parameter"
1 file changed, 5 insertions(+), 21 deletions(-)
```

Git 将会要求用户输入注释说明信息，主要用来解释用户还原该提交的原因，存在什么问题，为什么需要还原而不是进行修复。系统默认会采用被还原提交的 SHA-1 码：

```
$ git show --stat
commit 76d9e259db23d67982c50ec3e6f371db3ec9efc2
Author: Alice Developer <alice@example.com>
Date:    Tue Jun 16 02:33:54 2015 +0200
Revert "Support optional <count> parameter"
```

```
This reverts commit bb71a804f9686c4bada861b3fcd3cfb5600d2a47.
```

```
src/rand.c | 26 +++++-----
1 file changed, 5 insertions(+), 21 deletions(-)
```

常见的做法是只留下主题信息（方便查找还原记录），不过会用为何执行该还原操作的原因的描述信息替换其中的内容。

## 还原错误合并

有时用户可能需要还原一个合并操作。假定用户已经合并了变更，但是事后证明它们被合并的时机过早，该合并会导致程序功能退化。

假定用户合并的分支是一个主题分支，现在已经把它合并到 **master** 分支上了，如图 8-5 所示。

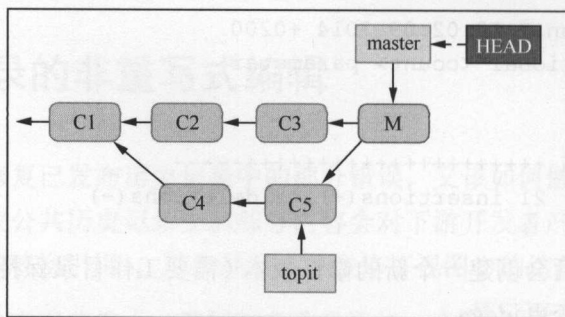


图 8-5 一个偶然的或者过早的合并提交，从起点开始还原合并和还原合并的重做

如果用户在发布该合并提交之前已经发现了该错误，而且有问题的合并提交只存在于本地版本库，则最简单的办法是使用 `git reset --hard HEAD^` 命令将它删除（详情可以参考第 4 章）。

如果用户发现合并提交有问题时为时已晚，例如在 `maser` 分支上，其后已经创建了多个提交并且已经被发布了，则一种办法是还原该合并提交。

不过如果合并提交存在多个父提交，即拥有多个变更集的话就比较难办了。为了还原一个合并提交，用户需要声明希望还原哪一个补丁，换句话说，就是指定某个父提交作为主干。在这种特殊场景中，假定合并提交之后存在多个提交对象（例如合并提交之后还有两个提交对象），相关的命令如下：

```
$ git revert -m 1 HEAD^^
[master b2d820c] Revert "Merge branch 'topic'"
```

这种合并还原的情况如图 8-6 所示。

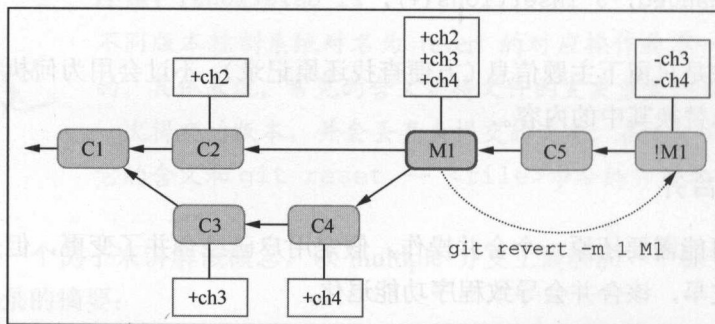


图 8-6 上图是 `git revert -m 1 <merge commit>` 命令之后的历史记录。被方框标记的特定提交表示它们是以类 diff 格式表示的变更集（专门用于合并提交的组合 diff 格式）

用户可以从新的 !M1 提交继续开始工作，就像该合并操作从未发生一样（!M1 用来表示提交 M1 的反向提交），只是从变更方面来说是这样的。

### 覆盖已还原合并

现在假定用户将在某个被还原了合并提交的分支上继续工作。也许是因为合并过早了，但是这并不代表开发工作也终止了。如果用户继续在同一分支上继续工作的话，也许需要创建一个包含错误修复的提交，将来某一天用户能够再次将它们正确地合并到主干分支上。又或者主干已经足够成熟，可以进行相关提交的合并了。如果用户只是简单地再次合并分支，那么麻烦可能会像上次一样出现在用户的面前。

意外的结果是 Git 只添加了自被还原提交以来的变更记录。变更是由旁边分支上的提交带来的，还原合并操作相关的变更并不在这里（见图 8-7）。换句话说，用户得到了一个奇怪的结果：在合并被还原之前，新的合并中并不包含用户自身分支上创建的变更（而是旁边分支上的）。

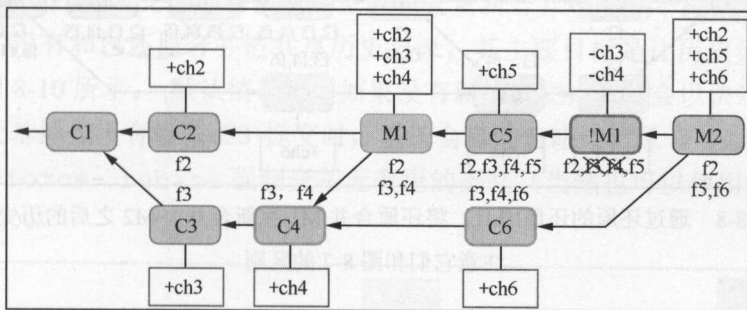


图 8-7 试图简单地在历史记录中对一个错误的合并提交执行还原合并重做导致了意外的错误结果。提交对象旁边的文本信息表示提交内部或者是从提交中提取的一组功能特性。边框加粗表示的 3 个提交是被合并的提交（“ours”和“theirs”的版本）和合并基底，即它们共同的祖先提交（“base”）

这是由还原操作撤销了变更（数据），但是没有撤销历史记录（DAG 视图上的修订）这样一个事实导致的。这说明新的合并提交将 C4 当作了一个共同的祖先提交，而它是旁边分支还原合并操作之前的一个提交。因为默认的三路合并策略只能看到 ours、theirs 和 base 3 个对象的快照的状态，它并没有通过历史记录查找相关的还原提交。

看上去共同的祖先提交 C4 和合并分支（即 theirs）C6 包含了提交 C3 和 C4 提供的特性，同时被合并的分支（ours）因为还原操作的缘故并没有包含它们。

对于合并策略来说，它们看上去就像从一个分支上删除了点东西，这意味着这种变更（删除）就是合并的结果（就像它们只对旁边的分支产生了影响）。具体来说，就像基底包



含一个特性，旁边的分支包含另外一个特性，但是当前的分支却不是这样（因为还原操作的缘故），因此结果中并没有包含它们。详情可以参考第7章。

有多种方法来解决这个问题，并且让 Git 可以正确地重新合并主题分支，即将特性 f3 和 f4 添加到结果中。选择哪种方案需要根据实际情况来决定，例如被合并分支是否已经被发布了。一般来说用户不会经常发布主题分支到公共版本库；如果用户经常这么做，也许通过包含合并所有主题分支的建议更新分支是个好办法，用户应该知道这些变更可能会被重写。

用于找回被删除的变更的一种解决方案是通过还原上一次的还原。操作结果如图 8-8 所示。在这种情况下，用户可以将获得的变更记录与在案的修订历史进行匹配。

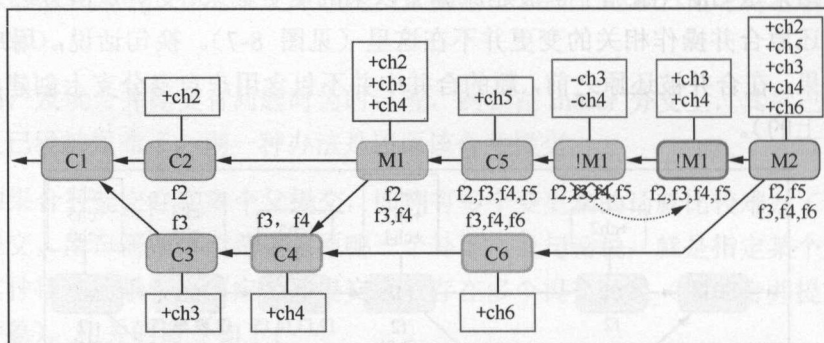


图 8-8 通过还原的还原!!M1，将还原合并 M1 重新合并为 M2 之后的历史。

注意它们和图 8-7 的区别

另外一种方案是改变历史修订的视图（也许是临时的），例如使用 `git replace` 命令对它进行编辑，将合并提交 !M1 改成一个非合并提交。上述两种方案都适用于至少部分分支被合并的情况，即 `topic` 分支被发布。

如果被合并提交中包含一些 bug（在 `topic` 分支上），并且分支已经被合并但是还未发布。如前所述，用户可以使用交互式变基修复这些提交。变基会修改历史记录，因此如果用户非常确信通过变基产生的新的历史记录和旧的历史记录，包括上述失败的合并提交以及还原提交之间不存在任何相同的修订版本，那么重新合并主题分支应该问题不大。

一般来说，用户将要变基的是一个主题分支，这里的示例中是 `topic` 分支；在分支的当前状态之上提取自 `master` 分支。这样一来用户的变更就能和当前工作进度保持同步了，后续的合并工作也更容易。现在主题分支上新增了变更记录，只需要将它再次合并到 `master` 分支即可，如图 8-9 所示。操作简单并且不会出现异常。

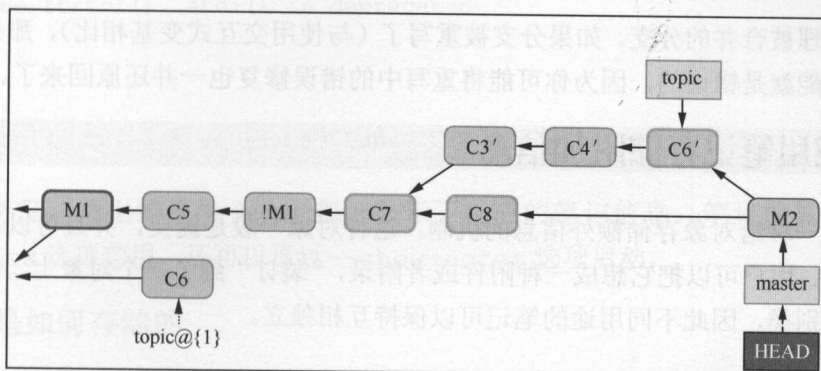


图 8-9 重新合并已变基分支后的历史记录，其中包括合并还原提交。其余未显示的历史记录和图 8-6 类似。边框加粗的 3 个节点代表合并提交（“ours”和“theirs”版本）以及新合并的基底，即共同的祖先提交（“base”）

更复杂的情况可能是当主题分支因为某些原因需要保留它的基底（例如也支持能够被合并到 maint 分支）。变基之后貌似对主题分支重新合并难度并不大，不过我们需要确保分支变基后没有和已还原合并链共享历史记录。其主要目标是让历史记录限定在一定范围内，如图 8-10 所示。默认情况下，如果没有新增变更，变基会以快进式处理修订记录（例如当变基操作没有修改 C3 提交时，系统会自动忽略该对象），因此我们需要使用选项 `-f / --force-rebase` 强制变基无变更的提交（当然也可以使用选项 `--no-ff`，它们是等效的）。

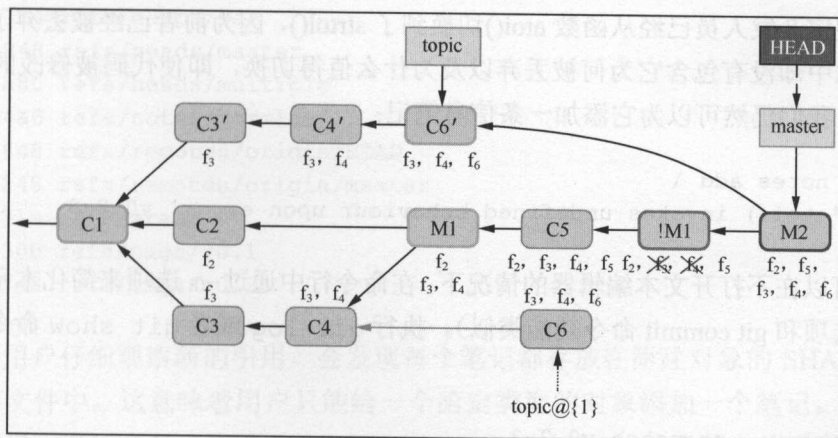


图 8-10 重新合并一个“适当位置”的已变基主题分支之后的历史，即还原预变基合并所在位置。用来标记提交的符号和图 8-7 相同

因此用户不应该盲目地还原一个合并还原。如何处理合并还原之后的再合并取决于用

户想如何处理被合并的分支。如果分支被重写了（与使用交互式变基相比），那么还原这个合并还原可能就是错误的，因为你可能将重写中的错误修复也一并还原回来了。

### 8.3.2 使用笔记存储附加信息

笔记是一种给对象存储额外信息的机制，这种对象一般是提交，并且可以不破坏对象的内部结构。用户可以把它想成一种附件或者附录，“装订”到了一个对象上。每个笔记都进行了分门别类，因此不同用途的笔记可以保持互相独立。

#### 给提交添加笔记

有时用户希望给一个提交添加一些额外的信息——一个只有当它被创建之后才有效的信息。它可能是一个在上述提交中发现 bug 的描述性笔记，又或者在将来某个时间被修复的日志笔记（功能退化的情况下）。也可能是提交发布之后，用户才发现忘记给它添加一些重要的注释信息了，例如为何要创建该提交。或者用户发现还有另外一种解决方案，现在希望添加一些备忘信息，并能够和其他开发人员共享这一思路。

因为 Git 中的历史记录是不可变的；除非通过重写，否则无法修改它们（创建一个经过修改的副本，然后丢弃原来的版本）。历史记录的不可变性非常重要：它允许用户修订签名并信任它们，一旦通过校验，历史记录就无法被修改。用户能够做的只是通过笔记给它添加额外的信息。

现在假定开发人员已经从函数 `atoi()` 切换到了 `strtol()`，因为前者已经被丢弃了。但是提交注释信息中却没有包含它为何被丢弃以及为什么值得切换，即使代码被修改的时间已经比较长了，我们仍然可以为它添加一条信息笔记：

```
$ git notes add \  
-m 'atoi() invokes undefined behaviour upon error' v0.2~3
```

用户可以在不打开文本编辑器的情况下，在命令行中通过 `-m` 选项来简化本示例的注释流程（该选项和 `git commit` 命令功能类似）。执行 `git log` 或者 `git show` 命令后将会看到该笔记：

```
$ git show --no-patch v0.2~3  
commit 8c4ceca59d7402fb24a672c624b7ad816cf04e08  
Author: Bob Hacker <bob@company.com>  
Date: Sun Jun 1 01:46:19 2014 +0200
```



Use `strtoul()`, `atoi()` is deprecated

Notes:

`atoi()` invokes undefined behaviour upon error

如你所见，输出结果中 **Notes:** 部分显示了对应的笔记信息。笔记的显示可以通过 `--no-notes` 选项禁用，还可以通过 `--show-notes` 选项启动。

## 笔记是如何存储的

在 Git 中，笔记是用位于 `refs/notes/` 命名空间下另外的引用存储的。默认情况下，提交的笔记是采用 `refs/notes/commits` 中的引用存储的；用户可以通过配置变量 `core.notesRef` 修改这一设置，该变量继而也可以通过环境变量 `GIT_NOTES_REF` 进行覆盖。

任何一个变量的值都必须是完全合法的（即它必须包含 `refs/notes/prefix`，但这一要求在最新版的 Git 中得到了放宽）。如果给定引用不存在，系统并不会报错，不过这意味着没有可供显示的笔记。这些变量决定了提交注释信息中 **Notes:** 行显示的笔记类型和 `git notes add` 命令去哪里访问相关的笔记。

用户可以在版本库发现新的引用类型：

```
$ git show-ref -abbrev
2b953b4 refs/heads/bar
5d25848 refs/heads/master
bb71a80 refs/heads/multiple
fcac4a6 refs/notes/commits
5d25848 refs/remotes/origin/HEAD
5d25848 refs/remotes/origin/master
b35871a refs/stash
995a30b refs/tags/v0.1
ee2d7a2 refs/tags/v0.2
```

如果用户仔细观察新的引用，会发现每个笔记都存放在附注对象的 SHA-1 标识符之后的具名文件中。这意味着用户只能给一个给定类型的对象添加一个笔记。用户总是可以编辑笔记，追加内容（使用 `git notes append` 命令），替换其中的内容（使用 `git notes add --force` 命令）。在交互模式下，Git 会用文本编辑器打开笔记中的内容，因此 `edit/append/replace` 在这里的功能是类似的。和提交的不同之处在于，笔记是可变的：



```
$ git show refs/notes/commits
commit fcac4a649d2458ba8417a6bbb845da4000bbfa10
Author: Alice Developer <alice@example.com>
Date:   Tue Jun 16 19:48:37 2015 +0200
```

```
Notes added by 'git notes add'
```

```
diff --git a/8c4ceca59d7402fb24a672c624b7ad816cf04e08 b/8c4ceca59d7402fb2
4a672c624b7ad816cf04e08
new file mode 100644
index 00000000..a033550
--- /dev/null
+++ b/8c4ceca59d7402fb24a672c624b7ad816cf04e08
@@ -0,0 +1 @@
+atoi() invokes undefined behaviour upon error
```

```
$ git log -1 --oneline 8c4ceca59d7402fb24a672c624b7ad816cf04e08
8c4ceca Use strtol(), atoi() is deprecated
```

提交相关笔记是存储在独立的历史时间线（元数据）上的，不过这对于其他类型的笔记影响不大：笔记引用可以直接指向树对象，而不是 refs/notes/commits 这样与提交对象有关的目录。

书籍或者博客上经常会忽略一个非常重要的问题，即包含笔记内容的文件对应的完整路径，它并不是文件的基础名称，它是标识被附加了笔记的对象的。如果笔记数量很大，Git 会采用扇出式目录结构来存储它们，例如会把前文所述的笔记存放在 8c/4c/eca59d7402fb24a672c624b7ad816cf04e08 目录下面（注意斜线）。

## 其他笔记的用途

通常笔记用于给提交对象添加额外信息。不过即使这些笔记只有附加到提交对象上才会有意义，但是至少在某些情况下，可以使用不同种类的笔记存储不同信息。它可以用于处理个人信息，能够决定显示哪些部分，哪些部分推送到公共版本库，同时还支持查询特定部分的个人信息。

为了在命名空间中创建一个有别于默认类型的笔记（默认是指 notes/commits，或者 core.notesRef 指定的类型），用户需要在添加笔记时声明笔记的类别：

```
$ git notes --ref=issues add -m '#2' v0.2~3
```

现在，默认情况下，Git 只会在提交注释信息之后显示 `core.notesRef` 变量指定的类型的笔记。为了将其他类型的笔记添加进来，用户必须使用 `git log -notes=<category>` 命令指定显示的类型(<category>是一个合法或者不合法的引用名或通配符，用户可以使用 `--notes=*` 显示所有笔记类型)，或者在默认的配置变量 `isplay.notesRef` 中配置需要显示的笔记类型（或者使用环境变量 `GIT_NOTES_DISPLAY_REF` 进行配置）。用户可以声明配置变量多次，例如 `remote.<remote-name>.push`（或者可以像使用环境变量那样声明一组由分号分隔的参数列表），也可以声明一个全局的匹配模式：

```
$ git config notes.displayRef 'refs/notes/*'
$ git log -1 v0.2~3
commit 8c4ceca59d7402fb24a672c624b7ad816cf04e08
Author: Bob Hacker <bob@company.com>
Date: Sun Jun 1 01:46:19 2014 +0200
```

```
Use strtol(), atoi() is deprecated
```

```
Notes:
```

```
atoi() invokes undefined behaviour upon error
```

```
Notes (issues):
```

```
#2
```

笔记的用途还有很多。例如可以使用笔记可靠地标记哪些补丁（提交）属于上游（向前转发到开发分支）或者下游（向后转发到更稳定的分支或版本库），即使上/下游的版本不一致。如果补丁还不成熟，不能发送到上游或者下游，还可以将它标记为延迟发送。

如果需要手工输入的话，那么它比使用 `git patch-id` 命令检测是否存在变更集的机制更可靠（该命令还可以用于变基，`git cherry` 命令，或者 `git log` 命令搭配 `--cherry /--cherry-pick /--cherry-mark` 选项一起使用）。当然，这种情况属于用户不是从主题分支开始工作，而是从拣选提交开始工作。

笔记还可以用来存储后续提交（但是处于预合并状态）的代码审计结果，并且可以提醒其他开发人员采用该版本的具体原因。

笔记还可以用来标记 bug、bug 修复提交、修复验证等。用户经常会在很久之前就已经发布的提交中发现 bug，这也是需要为它添加笔记的原因。如果用户在提交公开发布之前就发现了 bug，那么可以对该提交进行重写。

在这种情况下，首先如果 bug 已经被提交报告了，并且它会影响程序功能，那么用户需要定位引入该 bug 的修订版本（例如使用第 2 章介绍的 `git bisect` 命令）。然后用户将会希望对该提交进行编辑，把这个 bug 实体在项目问题跟踪系统中的标识符添加到 bugs、defects 或者 issues 等类别的笔记中（通常会使用数字或者特定的前缀对它们进行命名，例如 bug: 1385）。也许用户还会希望将该 bug 的描述信息包含其中。如果该 bug 涉及安全问题，它可能会被分配一个漏洞标识符，例如通用漏洞披露(CVE)代码，这类信息会被存放到 CVE-IDs 类的笔记中。

然后过了一段时间，幸运的是，该 bug 被修复了。只需要将上述标记存在 bug 的提交那样，对该 bug 附加该 bug 已被修复的信息即可。例如在 `refs/notes/fixes` 目录下的笔记。如果不走运，开发人员第一次尝试修复该 bug 后没有成功，那么用户就必须对该修复笔记进行编辑，或者为该修复再创建一个修复提交。如果用户正在使用 `bugfix` 或者 `hotfix` 分支（专门用于处理主题分支的 bug 修复分支），如第 6 章所述，那么将会很容易通过合并上述 `bugfix` 分支找到它们，并对它们进行整合。如果用户不擅此道，那么使用笔记附注修复记录，使用拣选提交创建一个增补性的提交会是一个不错的解决方案。例如通过将笔记添加到 `alsoCherryPick` 或者 `seeAlso` 类目下，以及用户希望添加的类目名下。也许一个最初的提交者或者问答讨论组能够正确地修复和测试该 bug；最好能够在提交发布之前对它进行测试，不过并不是总能如愿，因此 `refs/notes/tests` 就可以发挥作用了。

第三方工具可以使用笔记存储每个提交的特定信息。例如 `Gerrit`，它是一款免费的、基于 Web 的团队代码协作工具，它把审核代码相关的信息存放在 `refs/notes/reviews` 目录下。其中包括 `Gerrit` 用户提交变更时的名字和电子邮件地址，修订被提交的时间，`Gerrit` 实例中代码审核的 URL 地址，审核标签和评分（还包括审核者的标识符），项目和分支的名称等：

Notes (review):

```
Code-Review+2: John Reviewer <john@company.com>
Verified+1: Jenkins
Submitted-by: Bob Developer <bob@company.com>
Submitted-at: Thu, 20 Oct 2014 20:11:16 +0100
Reviewed-on: http://localhost:9080/7
Project: common/random
Branch: refs/heads/master
```



类似的工具还有 `git svn`，它是能够在 Subversion 版本库和 Git 版本库直接进行双向操作的 Subversion 旁客户端，可以使用笔记存储原生的 Subversion 标识符，而不是在提交注释中追加信息（或者将它们一并删除）。

在更复杂的应用示例中，用户可以使用笔记机制存储编译结果（也可以是存档、安装包或者是可执行程序）、提交或者标签绑定等。理论上来说，用户可以将编译结果存放在标签中，但是用户一般会希望标签里面包含完美隐私（Pretty Good Privacy, PGP）的签名，或者还包括发布版信息高亮的支持。当然，在大部分情况下用户都会希望获取所有标签，但是并不是每个人都在意占用硬盘空间为预编译程序提供便利。用户可以逐一决定是否获取给定类别的笔记（例如可以忽略预编译的二进制文件），同时自动关联标签。这也是笔记优于标签的地方。

本示例将会演示如何正确地创建一个二进制笔记。用户可以使用如下技巧安全地创建一个二进制笔记：

```
# store binary note as a blob object in the repository
$ blob=$(git hash-object -w ./a.out)
# take the given blob object as the note message
$ git notes --ref=built add --allow-empty -C "$blob" HEAD
```

用户不能简单地使用 `-F ./a.out`，因为这不是二进制安全的——提交将会被剥离（其中的内容会被误当作注释，即以#开头的行）。

笔记机制也被用作 `textconv` 过滤器存放缓存（详情可以参考第 4 章）。用户需要做的只是配置该过滤器，将它的 `cachetextconv` 设置为 `true`：

```
[diff "jpeg"]
  textconv = exif
  cachetextconv = true
```

这里 `refs/notes/textconv/jpeg` 目录下的笔记用来绑定转换成 `blob` 对象的文本（名称在过滤器后面）。

## 重写历史记录和笔记

笔记可以绑定到对象的附注中，一般来说这种对象就是提交，通过它们的 SHA-1 标识符实现。那么用户在重写历史记录时，笔记又意味什么呢？在新的被重写历史记录中，对象的 SHA-1 标识符一般来说都是不一样的。

事实证明，用户可以对它们进行灵活、广泛的配置。首先，用户可以通过多值配置变



量 `notes.rewriteRef` 指定在附注对象被重写过程中可以拷贝哪些类别的笔记。这个设置可以被包含的由分号分隔的、一组完全合法的笔记引用列表的环境变量 `GIT_NOTES_REWRITE_REF` 覆盖，并且支持通配符引用模式匹配。这个设置没有默认值，用户必须配置这个变量来启用重写。

其次，用户可以配置是否在重写过程中拷贝笔记，不过这需要依赖重写过程中具体的命令类型（当前支持将 `rebase` 和 `amend` 作为命令的值进行传递）。这也可以通过布尔值变量 `notes.rewrite.<command>` 进行设置。

此外，用户还可以在目标提交已经包含一个笔记的情况下，在重写过程中决定该如何处理拷贝笔记，例如在通过交互式变基压缩提交时。用户必须通过配置变量 `notes.rewriteMode` 或者环境变量 `GIT_NOTES_REWRITE_MODE` 在覆盖（选用追加提交中的笔记）、串联（默认值）和忽略（选用原来提交的笔记）三者之间做出选择。

## 发布和检索笔记

现在已经可以在本地版本库添加笔记了，那么如果用户希望将笔记和其他人共享又该怎么办呢？如何将它们发布到公共版本库上？用户和其他开发人员又如何从其他版本库中获取笔记？

这里我们可以回顾一下刚学过的 Git 知识。8.3.2 节“笔记是如何存储的”部分曾经提到过，笔记是存储在版本库对象数据库中的，并且使用了特殊的引用存放在 `refs/notes/` 命名空间之下。

笔记的内容是以 `blob` 对象存储的，通过上述特殊的引用进行编址。提交笔记（位于 `refs/notes/commits` 中的笔记）存储的是笔记的历史，因此 Git 也允许用户以无历史的形式存储笔记。因此，用户需要做的是获得笔记的引用，笔记的内容也就随之而来了。这也是常见的版本库同步机制（对象迁移）。

这意味着如果希望发布笔记，用户需要在相应的远程版本库配置中配置相应的推送线（详情可以参考第 5 章）。假定用户正在使用一个独立的公共远程版本库（如果用户是维护人员，可能只会简单地使用 `origin` 库），也许用户会将它通过 `remote.pushDefault` 设置为默认的推送库，而且用户可以将笔记推送到任意目录，那么可以执行如下命令：

```
$ git config --add remote.public.push '+refs/notes/*:refs/notes/*'
```

在这种情况下，当 `push.default` 设置为匹配模式时（或者 Git 的版本太旧，默认会采用这种模式），也可能“push”线使用了特殊的 refspec “:” 或者 “+:”。总之，现在的条件已

经足够首次推送笔记了，后续过程中它们每次都会被自动推送：

```
$ git push origin 'refs/notes/*'
```

获取笔记稍微有一点复杂。如果用户自身没有生成特定类型的笔记，那么可以通过类镜像模式获取同名引用的笔记：

```
$ git config --add remote.origin.fetch '+refs/notes/*:refs/notes/*'
```

不过如果存在冲突的话，那么用户就需要先获取远程版本库上远程跟踪笔记的引用，然后使用 `git notes merge` 命令将它们和自己的笔记进行集成，详情可以参考帮助文档。



如果用户希望能够更便捷地合并 git 的笔记，甚至让它自动化，那么请遵循下列关键的约定：删除笔记内容中重复的单行条目值将会大有裨益。

目前远程跟踪笔记引用并没有规范的命名约定。不过用户既可以使用 `refs/notes/origin/*`（因此笔记简化版的类目提交对应的远程版本库上 `origin` 的目录是 `origin/commits`），也可以从远程的原生版本库 `refs/remotes/origin/refs/*` 拉取 `refs/*` 目录下的所有内容（因此相应的提交目录就是 `refs/remotes/origin/refs/notes/commits`）。

### 8.3.3 置换机制应用

类替换/置换机制原本是为了能够将两个不同版本库的历史记录串联。

最初的目的就是通过创建两个版本库，来达到从其他版本控制系统切换到 Git 的目的：一个处理当前的工作，在空版本库中从最新的版本开始工作；另外一个用来处理历史数据，存储从原来系统中迁移过来的数据。通过这种方式，能够花时间对历史数据进行忠实的转换，如果转换出现了问题，还可以及时对它进行修复，同时不会影响当前的开发工作。

我们需要的是将两个版本库历史串联起来的某种机制，能够回溯项目的完整历史（例如对于 `git blame` 命令，是行历史注释信息）。

#### 置换机制

目前流行的这类工具就是替换（置换）机制的化身。通过它们，用户可以替换任何对

象,甚至可以创建一个覆盖并基于此创建一个虚拟的历史记录(版本库的虚拟对象数据库),以便 Git 中大部分命令可以返回一个置换对象来替代原有的对象。

但是原有的对象仍然存在,采用了置换机制后的 Git 在行为上可以消除数据丢失的可能性。用户可以对 git 命令使用 `--no-replace-objects` 选项或者环境变量 `GIT_NO_REPLACE_OBJECTS` 访问 Git 包装器,以便获取它的原生视图。例如用户为了查看原生历史记录,可以使用 `git --no-replace-objects log` 命令。

置换的相关信息是存放在版本库中 `refs/replace/`命名空间下被置换对象的 SHA-1 码之后的引用名组成的,置换对象唯一的内容就是 SHA-1 码。不过不需要用户通过手工或者底层命令编辑它;可以通过命令 `git replace` 实现上述目的。

如前所述,除非特别声明,几乎所有命令都采用了置换机制。唯一的例外是可达性分析命令;这意味着 Git 将不会移除被替换对象,因为我们选择了置换对象,被置换对象已经不可达了。当然,置换对象可以通过置换引用继续访问。用户可以使用任意对象替换希望被替换的对象。当然在改变对象类型时,需要使用 `git replace -f <object> <replacement>` 命令明确告知 Git 系统用户自己到底在做什么。还因为这样一种变化会导致 Git 出现故障,它被要求访问的对象和返回的对象类型不一致。

使用 `git replace --edit <object>` 命令,用户可以交互式地编辑它的内容。实际情况是,Git 打开对象内容并对它进行编辑,编辑完毕之后,Git 创建了一个新对象并替换了原有对象的引用。对象格式(具体来说就是提交对象的格式,用户经常编辑的就是提交对象)如本章前文所述。用户可以修改提交注释信息、父提交、作者信息等。

### 示例——使用 git replace 串联历史记录

现在假定用户希望将版本库一分为二,也许是因为性能方面的考虑,同时用户又希望可以像访问一个版本库历史记录那样处理上述两个版本库的历史记录。又或者它们是变更某个 SCM 后被一分为二的历史记录,新建的版本库处理当前的开发工作(使用一个空的历史记录从当前项目状态启动切换后的开发工作),并且和转换后的历史版本库保持隔离。

如何分割历史记录已经在本章 `git filter-branch` 命令的应用示例中介绍过。一种解决方案是在用户希望分割的提交对象上执行 `git replace --graft<to be root>` 命令,然后使用不带过滤器的 `git filter-branch -- --all` 命令对历史记录进行永久性的分割。

大部分情况下,用户也许会希望在历史版本库的基础上创建某类信息,例如添加 README 文件,提醒用户如何找到当前正在工作的版本库。为了简单起见,图 8-11 中并



没有显示这类提交。

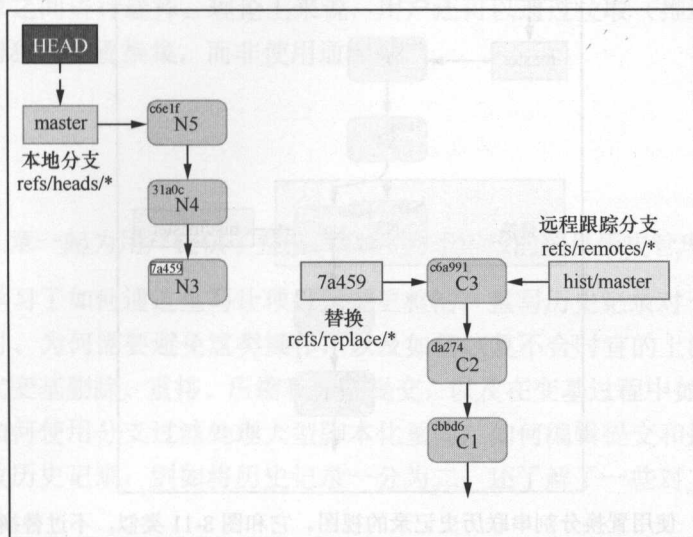


图 8-11 禁用置换机制的分割历史视图 (`git --no-replace-objects`)。左上角提交的 SHA-1 代码代表它们的标识符。注意，图中的 SHA-1 标识符都是用 5 位十六进制符号简写表示的

如何串联历史记录依赖于原有历史记录是分割还是串联的。如果原有历史记录是串联的，那么分割它，只需要告诉 Git 使用 `git replace <post-split> <pre-split>` 命令使用预分割版本替换后续分割版本。如果版本库原来是分割的，那么通过执行 `git replace` 命令搭配 `--edit` 或者 `--graft` 选项即可。

分割历史原本就已存在，只是在视图上被隐藏了。对于所有 Git 命令，历史记录和图 8-12 类似。如前文所述，用户可以禁用置换机制，在这种情况下，用户将看到的结果和图 8-11 类似。

## 历史笔记——grafts（移植）

首次尝试创建一种机制来串联若干历史记录的是移植技术（grafts）。它是一个包含受影响的提交和它们用空格分隔的置换父提交的简单文件 `.git/info/grafts`。

这种机制只适用于提交，并且只允许修改亲子关系。它不支持在 Git 内部传播这类信息。用户不能临时禁用移植机制，至少不是那么轻而易举。此外，它本质上是不安全的，因为对可达性检查命令也不例外，从而有可能导致 Git 在清理内存时（垃圾收集）发生意外，继而误删除有用的对象。



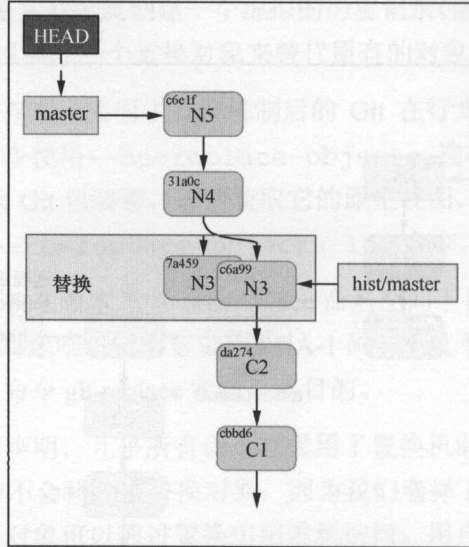



图 8-12 使用置换分割串联历史记录的视图。它和图 8-11 类似，不过置换引用的方式不同——因为结果是通过置换操作获得的

不过，用户可以在示例中找到它的具体应用。如今，它已经过时了，特别是 `git replace --graft` 命令的出现。如果用户想使用移植机制，可以考虑使用置换对象代替它们。Git 源码中有一个名为 `contrib/convert-grafts-to-replace-refs.sh` 的脚本可以帮助用户达到上述目的。

 浅克隆的管理是通过和移植机制的类似（`git clone --depth=<N>` 命令的执行结果，是简化版历史记录的克隆）。`.git/shallow` 文件实现的。该文件是由 Git 管理的，而非用户。

### 发布和检索置换集

如何发布置换集？如何从远程版本库获取它们？因为置换集采用引用访问，事情就非常简单了。

每个置换对象在命名空间 `refs/replaces/` 下都是一个独立的引用，因此用户可以通过通配符匹配拉取或者推送管道获取所有置换集：

```
+refs/replace/*:refs/replace/*
```

一个对象只对应了一个置换对象，因此合并置换集也很方便。用户只能在一个替换对象获取其他对象之间进行选择。理论上来说，用户还可以通过拉取（推送）独立的置换对象引用集合获得独立的置换集，而非使用通配符。

## 8.4 小结

本章和第 6 章一起为用户提供了整洁、可读、易于审核的项目历史管理所需的所有工具。

读者已经学习了如何通过重写让项目历史更整洁，重写历史记录对于 Git 系统来说意味着什么，何时、为何需要避免这类操作，以及如何恢复不合时宜的上游重写；同时还学习了通过交互式变基删除、重排、压缩和分割提交，以及在变基过程中如何测试每个提交。读者还了解了如何使用分支过滤处理大型脚本化重写，如何编辑提交和提交的元数据，如何永久性地修改历史记录，例如将历史记录一分为二；还了解了一些对上述操作有帮助的第三方外部工具。

读者还了解了如果无法重写历史时该如何处理，即如何通过创建包含相应变更的提交来修复错误（例如使用 `git revert` 命令），如何使用笔记给现有的提交附加额外信息，如何使用置换机制修改历史记录的虚拟视图；学习了如何恢复一个错误的合并和恢复合并之后的重合并；学习了如何拉取和发布笔记和置换对象。

为了切实理解历史重写，以及笔记和置换背后的机理，本章节介绍了 Git 内部的基础知识和底层命令的脚本应用（包括脚本重写）。

第 9 章将会介绍在版本库内部从子模块到子树连接不同子项目的多种方法。读者将会了解到在一个版本库中管理或者简化管理大型仓库的技术。将一个大型项目分割成子模块是方法之一，不过并不是解决此问题的唯一方案。

## 第 9 章

# 子项目管理——构建活动框架

第 5 章已经介绍过如何管理多个版本库，同时第 6 章向读者介绍了在这些版本库中使用多个分支和工作流水线进行软件开发的技术。目前为止，这些版本库都是以单个项目下的多个版本库出现的。不同项目之间都是独立开发的，其中的版本库也都是相对独立的。

本章将会通过多种方式介绍框架项目的单个版本库中不同子项目之间的交互，从嵌入项目代码、其他目录（子树）的强行插入，到项目之间的版本库嵌套（子模块）。读者将会学习如何把一个子项目添加到一个主项目中，如何更新超级项目和子项目的状态。本章还会介绍如何发布变更到上游，以及将它们反向移植到相应的项目，并推送到远程版本库。不同子项目管理技术之间各有利弊。

子模块有时也用于管理大型资源文件。本章还会介绍如何处理 Git 中大型二进制文件以及其他大型资源文件的替代性方案。

本章的主题包含以下几个部分。

- 管理库和框架的依赖。
- 依赖管理工具——Git 外部依赖管理。
- 将代码作为子树导入子项目。
- 子树合并，git-subtree 以及相关工具。
- 版本库嵌套：将子项目插入另外一个子项目。
- 子模块探幽：gitlinks、.gitmodules 和 .git 文件。
- 子树和子模块用例及其优劣比较。
- 替代性的第三方解决方案、工具/助手。
- Git 与大型文件。

## 9.1 管理库和框架的依赖

将一个外部项目串联到用户自身项目中的原因有很多。因为将一个项目（有时也称子项目或者模块）添加到另外一个项目（有时也称超级项目、主项目或者容器）的原因也不尽相同。视实际情况的不同添加项目的种类也会不同。它们各有利弊，而且最重要的一点是，用户必须有能力选择正确的解决方案解决自己的问题。

假定读者正在研发一个 Web 项目，Web 应用客户端采用的是 JavaScript 脚本（例如单页应用中用到的 AJAX 技术）。为了方便开发，用户可能会使用一些 JavaScript 库或者 Web 框架，例如 jQuery。

这类脚本库是一个独立的项目。用户应该会希望能够将它固定在某个已知的工作版本（为了避免该脚本库未来某个版本添加新特性之后导致用户自己的项目出问题），同时也希望可以审核代码并能够自动更新到新版本。也许用户还希望将符合自己需求的变更集成到脚本库中，并将相关变更推送到上游（当然，用户肯定会希望自己的客户能够使用包含自己成果的脚本库，即使该成果没有被原来的开发者接受）。可以想见，也许用户并不想发布对上述脚本库所做的定制和变更到上游，但是仍然希望可以使用它们。

这在 Git 中是完全可行的。添加子项目有两种常用的方案可供选择：使用子树合并策略将代码导入到项目中和使用子模块链接子项目。

子模块和子树的目标都是复用其他项目的代码，它们通常都有自己的版本库，需要把它们添加到用户自己版本库的工作目录树上。该目标通常得益于集中维护若干容器版本库的复用代码，而不必诉诸于笨拙、不可靠的手动维护（通常是复制和粘贴）。

有时它会比较复杂。很多公司比较常见的情况是使用大量的内部应用程序时，是在通用依赖库或者是在一组代码库上构建的。开发人员一般希望独立开发每个这样的应用程序，在使用时再将它们与其他程序一起整合。分支和合并，应用自己的变更和定制特性等操作都是在独立的 Git 版本库中进行的。如果用户想碰碰运气的话，也可以采用独立的整体式版本库，不过需要做好处理组织简化、依赖、跨项目变更和工具化等问题的准备。

不过，一个应用使用一个 Git 版本库的这种划分也并非完美无缺。如何处理通用库？每个应用都使用了一些特定版本库的程序库，因此用户需要对此特别留意。如果程序库发布了新的版本，用户需要测试新版的程序库是否能够和用户现有代码兼容，从而不会导致自己的应用崩溃。不过通用程序库一般不会作为独立组件进行开发，它的开发是由与之有关的项目需求驱动的。开发人员根据应用程序的需要为它添加新的功能特性。在



一定的时间点上，他们愿意推送程序库自身的变更，以便可以和其他开发人员共享，这也是因为开发人员希望能够减轻维护这些特性的负担（项目外补丁的维护成本要求他们这么做）。

接下来又该怎么办呢？本章将会介绍一些管理子项目的方法。对于每种技术，将会介绍如何添加子项目到超级项目，如何让它们及时更新，如果创建用户自己的变更，以及如何发布特定的变更到上游。



#### 注意：

所有方案都需要子项目的所有文件存放于超级项目的单个子目录下。目前的方案都不支持用户将子项目文件和其他文件混合，或者让它们占用一个以上的目录。

但是用户在管理子项目时，不管是子树、子模块、第三方工具也好，还是管理 Git 外部的依赖也好，最好保持特定超级项目中模块代码的独立性（至少使用外部的非版本控制配置来处理这些特性）。对特定超级项目的修改有违模块化和封装性原则，会给两个项目造成不必要的耦合。

### 9.1.1 Git 外部依赖管理

很多情况下，技术背景（用户使用的技术栈）为用户使用打包和依赖管理提供了良好的支持。如果可能的话，这一般是优先选择的路径。它可以让用户更好地分割代码基，避免子模块和子树解决方案带来的副作用、并发症和不良影响（不同技术带来的复杂性）。它把版本控制系统从管理模块中移除，同时还可以从版本控制架构中受益，例如专门处理用户程序依赖的语义化版本控制（<http://semver.org/>）。

作为一个善意的提醒，下列是部分主流编程语言、开发栈和它们依赖管理/打包系统和注册工具的部分列表（完整列表在 <http://www.modulecounts.com/>）。

- Clojure 对应的工具是 Clojars。
- Go 对应的工具是 GoDoc。
- Haskell 对应的工具是 Hackage（注册）和 cabal（应用程序打包）。
- Java 对应的工具是 Maven Central（Maven 和 Gradle）。
- JavaScript 对应的工具有 npm（需要安装 Node.js）和 Bower。

- .NET 平台对应的工具有 NuGet。
- Objective-C 对应的工具有 CocoaPods。
- Perl 对应的工具有 CPAN (Comprehensive Perl Archive Network) 和 carton。
- PHP 对应的工具有 Composer、Packagist、PEAR 和 PECL。
- Python 对应的工具有 PyPI (Python Package Index) 和 pip。
- Ruby 对应的工具有 Bundler 和 RubyGems。
- Rust 对应的工具有 Crates。

有时, 这些工具未必能够满足需要。用户可能需要应用某些工作目录之外的补丁来对模块(子项目)进行个性化定制。但是因为某些原因, 用户不能将这些变更发布到上游使其公开。也可能这些变更只相对用户特定项目有效, 上游对建议变更的内容响应缓慢, 也可能是因为授权的原因, 也许有争议的子项目是一个内部模块、不方便公开, 它只允许在公司项目中使用。

对于上述情况, 用户需要自定义软件包注册(软件包版本库), 以便附加到默认的包管理器中方便调用, 或者需要将子项目作为私人软件包来管理, 这也是上述系统兼容的。如果无法兼容私人软件包, 那么诸如管理私人软件包的工具 Pinto 或者 Perl 语言的 CPAN::Mini 可能也是必需的。

## 9.1.2 手工导入项目代码

让我们来看看其中的一种解决方案: 为什么不能在项目中直接将软件包导入某些子项目中呢? 如果希望更新它们, 只需要拷贝包含新版本程序的文件即可。在这种方案中, 子项目代码是直接嵌套在超级项目代码中的。

最简单的方案就是每次用户希望将超级项目更新到新版本时, 只需使用新版本程序的文件将子项目目录中的内容覆盖。如果用户希望导入的项目并没有使用 Git, 或者它根本就没有使用版本控制系统, 也可能版本库是非公开的, 那么这将是唯一可行的解决方案。



### 将外源 VCS 版本库作为远程版本库

如果用户导入的项目代码采用的版本控制系统是 Git 以外的软件, 但是提供了很好的转换机制(例如使用了快速导入流), 那么可以通过远程版本库助手将外源



VCS 版本库设置为一个远程版本库（通过智能转换机制）。详情可以参考第5章和第10章的相关章节。

对于 Mercurial 和 Bazaar 版本库，用户可以使用 `git-remote-hg` 和 `git-remote-bzr` 版本库助手完成上述转换。

将导入的程序库更新到新版本也非常简单（原理也很容易理解）。移除相关目录下的所有文件，然后将程序库新版本文件添加到里面，例如从压缩归档文件中提取，然后在该项目下执行 `git add` 命令：

```
$ rm -rf mylib/  
$ git rm mylib  
$ tar -xzf /tmp/mylib-0.5.tar.gz  
$ mv mylib-0.5 mylib  
$ git add mylib  
$ git commit
```

一般情况下，该方法需要注意的事项有以下几点。

- 在 Git 中，用户的项目历史记录中只包含程序库导入时的版本。一方面，这使得用户的项目历史记录整洁易读；另一方面，用户没有细粒度访问项目历史记录的权限。例如，在使用 `git bisect` 命令时，用户只能断定问题是由更新程序库导致的，但是不能定位到程序库历史记录中具体某个导致异常的提交记录。
- 如果用户希望定制程序库代码，以便能够满足自身项目的实际需要，则在导入程序库新版本之后，用户需要以某种方式重新应用这些自定义变更。用户可以使用 `git diff` 命令提取这些变更（和导入时未变更版本进行比较），然后在升级程序库之后执行 `git apply` 命令；也可以使用变基或者交互式变基，以及某些补丁管理接口，详情可以参考第8章。Git 无法自动完成该任务。
- 每次导入程序库新版本时需要执行一系列特定命令来更新子项目：移除旧版本的文件，添加新版本文件，提交变更。即使可以使用脚本或者别名进行辅助，但是它不能像执行 `git pull` 命令那样容易。

### 9.1.3 包含子项目代码的 Git 子树

在一个更高级的解决方案中，用户可以使用子树合并将子项目历史记录与超级项目历



史记录串联。这在某些方面可能要比普通的 pull 操作复杂一些(至少导入子项目时是这样),不过它提供了一种自动化合并所有变更的方式。

根据用户的需求,这种方法有可能和用户的要求非常吻合,它包含以下几个优点。

- 用户将一直使用正确的程序库版本,永远不会因为意外而使用错误的版本。
- 方法简单易学,只使用标准的 Git 特性。如你所见,采用的都是最重要和最常用的操作,简单易学并且不容易出错。
- 用户应用程序的版本库永远是自包含的,因此克隆它时将会包含该程序所有必需的内容。这意味着该方法非常适合处理引用依赖的问题。
- 对用户版本库中的程序库应用补丁非常方便,即使用户没有上游版本库的提交权限。
- 在应用中创建一个分支的同时也会为程序库创建一个分支,其作用和分支切换是一样的。这也是用户希望的行为。在这一点上,它和子模块的行为(以及其他管理子项目的技术)形成了鲜明对比。
- 如果用户正在使用子树合并策略(详情可以参考第 7 章),例如使用 `git pull -s subtree` 命令,那么获取新版本的程序库将会和更新项目其他部分一样简单。

不过不幸的是,这种技术并不是完美无瑕的。对于很多用户和项目来说,这些缺点是无关紧要的,因为子树方法的简单性往往掩盖了它的不足。

下面是子树的不足之处。

- 每个使用程序库的应用程序相关文件数量也会加倍。目前还没有方便、安全的方法在不同项目和版本库之间共享它的对象,尽管存在共享 Git 对象数据库的可能性。
- 每个应用使用的程序库在工作区中都有对应的文件签出,即使用户可以通过稀疏签出(本章稍后会介绍)修改它们。
- 如果用户在应用中对引用的程序库拷贝进行了编辑,那么将这些变更发布和推送到上游并不是一件简单的事情。诸如 `git subtree` 或者 `git stree` 这样的第三方工具可以帮助用户达成目标。它们包含特定的子命令用于提取子项目的变更记录。
- 因为子项目文件和超级项目文件之间缺乏独立性,很容易将程序库的变更和应用的变更混在一个提交记录中。在这种情况下,用户可能需要重写历史记录(或者拷贝历史记录),详情可以参考第 8 章。

前两个问题说明子树并不善于管理包含备选依赖(只在某些特别功能中会用到)或者备选组件(例如主题、扩展和插件)的子项目,尤其是应用程序安装在文件系统中特定的



位置的情况。



### 使用轮替机制在副本之间共享对象

用户可以使用轮替机制减少版本库中的重复对象，换句话说，就是使用 `git clone --reference` 命令。不过在后续的垃圾收集过程中需要格外小心。有问题的部分是借用方版本库中部分历史记录存在的引用，但是在借出方版本库中却不存在对应的引用。轮替机制的详情可以参考第 11 章。

有多种方法可以用来管理子树导入的子项目。用户可以通过经典的 Git 命令，只使用相关的选项影响子项目，例如合并、拣选提交以及相关操作的 `--strategy=subtree`（或者默认递归合并策略的 `subtree` 选项 `--strategy-option=subtree=<path>`）。这方面的手动工作随处可见，大部分情况下都非常简单，并且为控制操作提供了最佳维度。不过这需要对其背后的机制有一个透彻的理解。

在现代 Git 软件中（自 1.7.11 版以来），`git subtree` 命令可以帮助用户处理二进制安装程序。它来自 `contrib/` 区域，并且没有完全集成（例如相关的说明文档部分）。这种脚本经过良好的测试并且非常健壮，但是它的一些观念比较奇特，容易让人感到迷惑，该命令并不支持全区间的子树操作。此外，这种工具只支持历史工作流的导入（稍后会解释该概念），某些内容在历史记录图形上的表示比较杂乱。

还有诸如 `git-stree` 这样的第三方工具可以辅助处理子树。

## 为子项目创建远程版本库

一般来说，在导入一个子项目时，用户希望能够方便地更新嵌入的文件，同时也希望可以继续和子项目进行交互。为此，用户需要将子项目（例如一个普通的程序库）设置为自己主项目（`super`）下的一个远程版本库引用，并能够访问它（`fetch`）：

```
$ git remote add mylib_repo https://git.example.com/mylib.git
$ git fetch mylib_repo
warning: no common commits
remote: Counting objects: 12, done.
remote: Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
From https://git.example.com/mylib.git
```

```
* [new branch]      master      -> mylib_repo/master
```

用户在查看远程跟踪分支 `mylib_repo/master` 时,既可以使用 `git checkout mylib_repo/master` 命令将它签出到脱离 HEAD 的状态,也可以在它之外创建一个本地分支,并使用 `git checkout -b mylib_branch mylib_repo/master` 命令签出该本地分支。

此外,用户可以使用 `git ls-tree -r --abbrev mylib_repo/master` 命令只显示它的文件列表,同时会发现子项目拥有一个和超级项目不同的项目根目录。另外需要注意的是其中的警告信息:没有共同提交 (`no common commits`),说明该远程跟踪分支包含的完整历史记录来自一个独立项目。

## 将子项目当作子树

如果用户没有使用 `git subtree` 这类特殊的工具,而是通过手动执行,那么接下来的步骤将会有一些复杂,并且会用到一些高级的 Git 概念和技术。幸运的是,这只需要执行一次即可。

首先,如果用户希望导入子项目历史记录,将需要创建一个合并提交来导入子项目。用户需要在超级项目的指定目录添加子项目文件。不幸的是,至少到编写本书时止,最新的 Git 版本使用 `-Xsubtree=mylib/` 策略选项并不能如用户所愿完成相关工作。用户必须分两步完成该工作:准备父提交,然后准备相关内容。

第一步是准备一个采用了 `ours` 合并策略的合并提交,不过不需要创建它(写入版本库)。该策略会串联历史记录,不过还会采用当前分支上当前版本的文件:

```
$ git merge --no-commit --strategy=ours mylib_repo/master
Automatic merge went well; stopped before committing as requested
```

如果用户只是希望像拷贝文件那样获取历史记录,那么可以跳过该步骤。

现在我们需要使用程序库对应的版本库中 `master` 分支上的内容更新我们的索引(提交对应的暂存区),并同时更新工作目录。所有这些操作都是在对应的子文件夹下进行的。用户还可以使用底层(管道)命令 `git read-tree` 达到上述目的:

```
$ git read-tree --prefix=mylib/ -u mylib_repo/master
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)
```

Changes to be committed:

```
new file:   mylib/README
[...]
```

如上所述，我们使用了-u 选项，因此工作目录能够和索引保持同步更新。



注意，千万不要忘记--prefix 选项中参数值里的斜杠。签出文件名字面上都添加了包含该参数值的前缀。

这些步骤在 Git 帮助文档的 HOWTO 章节有详细描述，其名为“[How to use the subtree merge strategy moved earlier](https://www.kernel.org/pub/software/scm/git/docs/howto/using-merge-subtree.html)（如何使用子树合并策略进行早期迁移）”：<https://www.kernel.org/pub/software/scm/git/docs/howto/using-merge-subtree.html>。

还有更方便的工具，如 git subtree 可供用户选择：

```
$ git subtree add --prefix=mylib mylib_repo master
git fetch mylib_repo master
Added dir 'mylib'
```

如果有必要，git subtree 命令会拉取子树的远程版本库；在手动解决方案中，用户不需要执行手动拉取的操作。

如果用户正在查看历史记录，例如使用 git log --oneline --graph --decorate 命令，将会发现该命令将应用程序（超级项目）的历史记录中的程序库相关的历史记录进行了合并。如果用户不希望如此，那么就很不走运了。git subtree 的子命令 add、pull 和 merge 支持的--squash 选项在这里并不能有所帮助。该工具比较特别的一点是，这个选项不会创建一个压缩合并，只会简单地合并经过压缩的子项目历史记录（就像是通过交互式变基压缩过一样），如图 9-2 所示。

如果用户不希望子树的历史记录和超级项目的历史记录有关联，那么可以考虑使用 git-stree 工具。该工具的另外一个优点是可以记住子树的设置，如果有必要，还能创建一个远程版本库：

```
$ git stree add mylib_repo -P mylib \
  https://git.example.com/mylib.git master
```

```
warning: no common commits
[master 5e28a71] [STree] Added stree 'mylib_repo' in mylib
 5 files changed, 32 insertions(+)
 create mode 100644 mylib/README
[...]
```

```
STree 'mylib_repo' configured, 1st injection committed.
```

和子树前缀（子目录）、分支等有关的信息是存放在本地配置 `stree.<name>` 组中的。这会止步于和 `git subtree` 命令的行为对比上，用户还需要为每个命令提供前缀参数。

## 使用子树克隆和更新超级项目

现在我们的项目中已经包含一个作为子树嵌入的程序库了，那么该如何调用它呢？因为子树背后的概念是只拥有一个版本库，即容器，用户可以方便地克隆这个版本库。

为了及时更新版本库，用户只需发起一个普通的 `pull` 请求，这将会同时更新超级项目（容器）和子项目（程序库）的内容。这和采用的方式、使用的工具，以及子树加入的形式都是无关的。它也是子树方法的一个特色之一。

## 使用子树合并获取子项目更新

自导入子项目以来，如果其中新增了一些变更，那么接下来会发生什么？更新嵌入超级项目的代码非常容易：

```
$ git pull --strategy subtree mylib_repo master
From https://git.example.com/mylib.git
 * branch          master      -> FETCH_HEAD
Merge made by the 'subtree' strategy.
```

用户可以先拉取更新，然后通过合并取而代之，这使得用户对代码的控制能力更强。或者可以使用变基代替合并，这也是一种行得通的办法。



在拉取子项目的同时不要忘记使用 `-s subtree` 选择合并策略。不过即使没有指定合并策略，合并操作仍然能够执行，因为 Git 的重命名检测机制通常可以发现文件从根目录（在子项目中）移动到了一个子目录（在超级项目中）。





有问题的情况是，子项目内部或者外部存在冲突文件。潜在的问题可能是由于 Makefile 文件和其他标准文件名导致的。

如果 Git 在检测正确的路径来执行合并操作时遇到问题，又或者用户需要使用普通合并策略中的某些高级特性（默认策略），则可以使用递归合并策略的子树选项 `-Xsubtree=<path/to/subproject>`。

用户还需要确保应用程序代码的其他部分能够和更新过的程序库完全兼容。

需要注意的是，采用此方案之后，用户子项目的历史记录会被附加到应用程序的历史记录中，如图 9-1 所示：

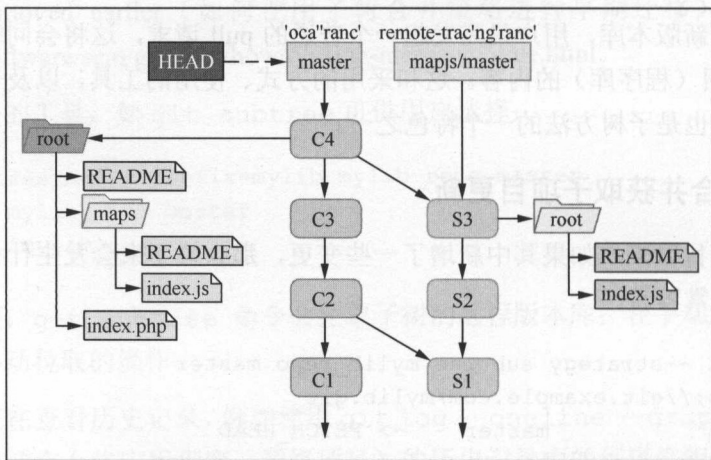


图 9-1 超级项目下包含子树合并子项目的历史记录

如果用户不希望子项目的历史记录和主项目的历史记录混在一起，那么可以在执行 `git merge` 命令（或者 `git pull` 命令）时搭配 `--squash` 选项一起使用，获得一个经过压缩的简化版历史记录。

```

$ git merge -s subtree --squash mylib_repo/master
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
$ git commit -m "Updated the library"

```

在这种情况下，用户将会获得只包含子项目变更的历史记录，不过这对于用户来说是利弊参半的。用户得到了更简单的历史记录，同时也把历史记录简化了不少。

通过 `git subtree` 命令和 `git stree` 工具，它们的 `pull` 子命令完全可以满足一般的需要了，并且自身还提供了子树合并策略。不过当前的 `git subtree pull` 命令需要用户重新指定 `--prefix` 和子树实体的设置。

注意，`git subtree` 命令总是会执行合并操作，即使添加了 `--squash` 选项；它只是在合并之前对提交记录进行简单的压缩（例如交互式变基过程中的 `squash` 指令）。反之，`git stree pull` 会一直压缩合并结果（例如 `git merge --squash`），它会将超级项目和子项目的历史记录隔离，继而达到不污染历史记录视图的目的，如图 9-2 所示。

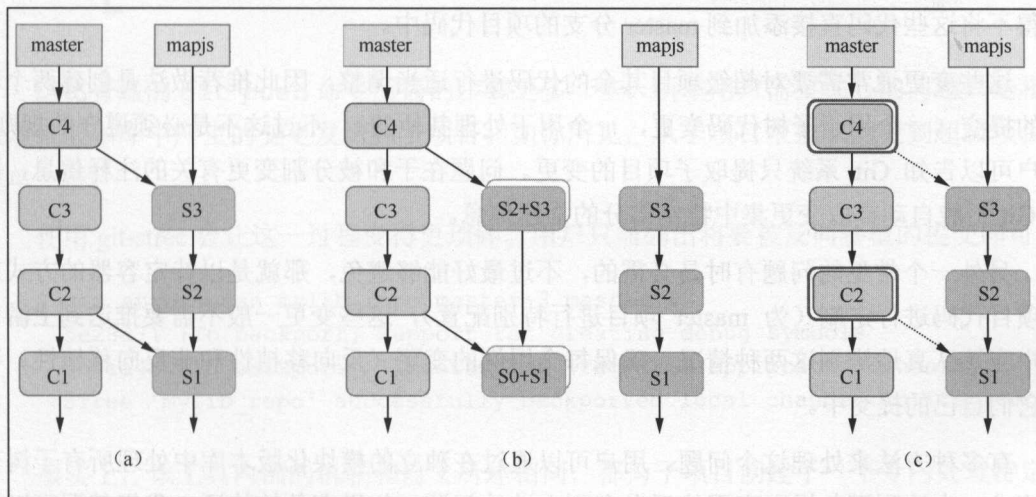


图 9-2 不同类型的子树合并：(a) 子树合并：`git pull -s subtree` 和 `git subtree pull`；(b) 压缩提交的子树合并：`git subtree pull --squash`；(c) 压缩过的子树合并：`git pull -s subtree --squash` 和 `git stree`。注意，(c) 中的虚线箭头表示提交 C2 和 C4 是如何创建的，而非它们的父提交

## 显示子树及其上游变更

为了比较子项目和当前工作目录版本之间的差异，用户需要在 `git diff` 命令中使用比较少见的查询语法。这是因为子项目（例如在 `mylib_repo/master` 远程跟踪分支上）的所有文件都在根目录下，同时也在超级项目的 `mylib/` 目录中（例如在 `master` 分支上）。我们需要指定子目录和 `master` 进行比较，并将它们放在修订标识符和冒号后面（忽略它们的话，系统会将之与超级项目的根目录进行比较）。

该命令如下所示：

```
$ git diff master:mylib mylib_repo/master
```

类似地，如果希望查看子树合并之后在 `mylib/` 目录下创建的提交与合并后的提交结果（即 `HEAD^2`）之间的差异，我们可以使用如下代码：

```
$ git diff HEAD:mylib HEAD^2
```

## 发布子树的变更到上游

在某些情况下，子项目的子树代码只能通过容器内的代码调用或测试；大部分主题和插件都有这样的约束。在这种情况下，在用户最终将子树代码反向移植到子项目上游之前，不得不将这些代码直接添加到 `master` 分支的项目代码中。

这些变更通常需要对超级项目其余的代码进行适当调整，因此推荐做法是创建两个独立的提交（一个用于子树代码变更，一个用于处理其他的），不过这不是必须遵守的规则。用户可以告知 `Git` 系统只提取子项目的变更。问题在于和被分割变更有关的注释信息，因为 `Git` 不能自动提取变更集中特定部分的描述信息。

另外一个常见的问题有时是必需的，不过最好能够避免，那就是以特定容器的方式对子项目代码进行定制（为 `master` 项目进行特别配置），这些变更一般不需要推送到上游。用户应该认真地辨别这两种情况，确保每个用例的变更（反向移植性和非反向移植性）都在它们自己的提交中。

有多种方法来处理这个问题。用户可以通过在独立的模块化版本库中处理所有子树变更请求，来达到避免提取变更然后发布到上游的问题。如果有条件的话，我们甚至可以要求首先将子项目变更发布到上游，然后只将通过上游验证的变更添加到容器中。

如果用户需要提取子树变更集，那么一种可行的办法是利用分支过滤命令 `git filter-branch --directory-filter`（或者 `--index-filter` 选项辅以相应的脚本）。另外一种简单的办法是只使用 `git subtree push` 命令。不过在反向移植每个提交时会接触到子树中有问题的内容。

如果用户希望发送到上游的变更只是子项目中需要集成到主项目版本库中的内容，那么解决方案要稍微复杂一些。一种办法是在子项目远程跟踪分支之外创建一个本地分支专门处理反向移植。提取自上述子树跟踪分支意味着它将子树作为根节点，并且只包含子模块文件。

子项目为了反向移植变更引入的分支，将会需要子项目上游版本库添加相应的分支作



为它的上游分支。通过上述配置，用户可以使用 `git cherry-pick -strategy=subtree` 选择感兴趣的提交，然后发送到子项目的上游。用户可以方便地使用 `git push` 命令将该分支添加到子项目的版本库中。



为了谨慎起见，最好声明 `--strategy=subtree` 选项，不过拣选提交没有它也能正常工作。这样做目的是确保能够完全忽略子项目文件之外的文件（子树之外）。这也可以用来从混合提交中提取子树的变更记录；如果没有声明该选项，Git 将会拒绝完成拣选提交操作。

这比普通的 `git push` 命令所需的步骤更多一些。所幸用户需要关心的问题只是将超级项目版本库中产生的变更发送到子项目。如你所见，从子项目中拉取变更到超级项目更简单一些。

使用 `git-stree` 会让这一过程变得更琐碎。用户只需列出将要被反向移植的提交即可：

```
$ git stree push mylib_repo master~3 master~1
5e28a71 [To backport] Support for creating debug symbols
5b0aa4b [To backport] Timestamping (requires application tweaks)
STree 'mylib_repo' successfully backported local changes to its remote
```

事实上，该工具内部的机制和前文所述相同，都为子项目创建了一个专门处理特定反向移植的本地分支。

### 9.1.4 子模块解决方案——版本库嵌套

从子项目导入代码到超级项目的子树方法也有不足之处。大部分情况下，子项目和其容器是两个完全不同的项目：用户的应用程序需要依赖程序库，但是它们明显是互相独立的实体。将它们的历史记录串联起来并不是一个好的解决方案。

此外，嵌入的代码和导入的子项目历史记录会一直存在于文件系统中。为此，子树技术并不是非常适合处理备选的依赖和组件（例如插件和主题）。它还不允许用户拥有对子项目历史记录的不同访问权限，这是为了避免在限制写入子项目时发生意外，可供用户选择的 Git 版本库管理方案有类似 `gitolite` 这样的工具（详情可以参考第 11 章）。

子模块方案能够将子项目代码及其历史记录在它自己的版本库中予以保留，并将版本



库嵌入超级项目的工作区，不过不会把它的文件当作超级项目的文件添加。

## Gitlinks、.git 文件和子模块命令

Git 中有一个名为 `git submodule` 的命令，它引入了子模块机制。不幸的是，驾驭它并不容易。为了正确地使用它，用户至少要对它的操作细节有一个基本的了解。这是两个风格迥异的组合：名为 `gitlinks` 的技术和 `git submodule` 工具本身。

在子树和子模块方案中，子项目都需要被包含到超级项目工作目录下特定的文件中。不过子项目的子树代码属于超级项目的版本库，而子模块则不是。通过使用子模块，每个子项目能够替换它容器版本库工作目录下自身的版本库。子模块代码拥有属于它自己的版本库，超级项目自身存储简单的元数据信息需要获取子项目文件的相应修订记录。

在实际工作中，时下流行的 Git 软件中的子模块会采用一个包含单个 `gitdir` 路径的 `.git` 文件，即包含到实际版本库文件夹的相对路径名。

子模块脚本库实际上位于超级项目的 `.git/modules` 文件夹中（并且可以通过 `core.worktree` 做相应的配置）。这么做的主要原因是为了应对超级项目中分支不包含任何子模块的情况。它可以避免在切换到不包含子模块的超级项目修订时，不得不对子模块版本库造成损害的情况出现。



用户可以将包含 `gitdir` 的 `.git` 文件看作是：`.git` 目录符号引用的等价物，一个操作系统独立的符号链接替代品。版本库的路径不一定必须是一个相对路径。

```
$ ls -aloF plugins/demo/
total 10
drwxr-xr-x 1 user    0 Jul 13 01:26 ./
drwxr-xr-x 1 user    0 Jul 13 01:26 ../
-rw-r--r-- 1 user  32 Jul 13 01:26 .git
-rw-r--r-- 1 user    9 Jul 13 01:26 README
[...]
$ cat plugins/demo/.git
gitdir: ../../.git/modules/plugins/demo
```

尽管如此，超级项目容器和子项目模块实际上就是独立的版本库：它们拥有自己的历史记录、自己的暂存区，以及自己的当前分支。因此用户在输入命令时务必小心看看自己是否在一个子模块中，因为命令的执行环境和影响会有显著的不同！

子模块背后的核心理念是超级项目的提交会记录子项目修订的精确信息；这个引用会使用子项目提交的 SHA-1 标识符。和其他依赖管理工具使用类 **manifest** 文件不同，子模块方案会使用名为 **gitlinks** 的机制将这些信息存储在一个树对象中。**Gitlink** 是一个从树对象（在超级项目版本库中）到提交对象（一般在子模块版本库中）的引用，如图 9-3 所示。

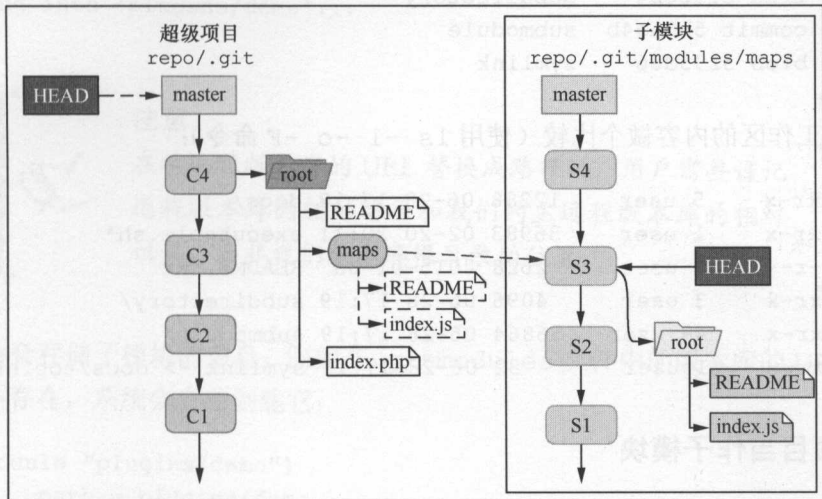


图 9-3 用子模块表示子项目的超级项目历史记录。左手边浅灰色阴影中的子模块文件表示以文件形式存在于超级项目工作目录的内容，但是它们在超级项目的版本库中并不存在

回想一下第 8 章中对版本数据库中对象类型的介绍，每个提交对象（表示项目的一个修订）都指向一个包含版本库内容快照的树对象。每个树对象引用的 **blob** 和树结构分别表示文件内容和文件目录。树对象是通过提交对象唯一的一组标记来引用的，其中包括提交对象关联修订的文件内容、文件名和文件权限。

还有一点务必记住，提交对象之间是互相关联的，并以此创建了修订的有向无环图（DAG）。每个提交对象引用了 0 个或者多个父提交，并一起组成了整个项目的历史记录。

前文所述的每种引用类型都是可达性检查的有机组成部分。如果对象的指针丢失了，那么意味着版本库被损坏了。

**gitlinks** 在这方面稍有不同。树对象中的实体指向的提交引用的对象来自其他独立的版本库，即名义上的子项目版本库（子模块）。实际上子模块提交不可达并没什么问题，这样一来用户就可以随意地包含子模块；没有子模块版本库，同时在 **gitlink** 中也就没有相关提交的引用。

对一个项目中所有类型的对象执行命令 `git ls-tree --abbrev HEAD` 的结果和

列内容类似：

```
040000 tree 573f464 docs
100755 blob f27adc2 executable.sh
100644 blob 1083735 README.txt
040000 tree ef9bcb4 subdirectory
160000 commit 5b0aa4b submodule
120000 blob 3295d66 symlink
```

将它和工作区的内容做个比较（使用 `ls -l -o -F` 命令）：

```
drwxr-xr-x 5 user 12288 06-28 17:18 docs/
-rwxr-xr-x 1 user 36983 02-20 20:11 executable.sh*
-rw-r--r-- 1 user 2628 2015-01-03 README.txt
drwxr-xr-x 3 user 4096 06-28 17:19 subdirectory/
drwxr-xr-x 48 user 36864 06-28 17:19 submodule/
lrwxrwxrwx 1 user 32 06-28 17:18 symlink -> docs/toc.html
```

## 将子项目当作子模块

使用子树时，第一个步骤通常是将一个子项目版本库作为远程版本库，这意味着子项目版本库的对象都是从子项目对象数据库拉取的。

使用子模块时，子项目版本库保持独立。用户可以在超级项目的工作区中手动管理子项目版本库的克隆，然后通过命令 `git add <submodule directory>`（注意尾部没有斜杠）添加 `gitlink` 引用。

### 重要提示！

一般来说，命令 `git add subdir` 和 `git add subdir/`（后者包含一个斜杠，这是遵循 POSIX 规范表示一个子目录的）是等效的。如果用户希望创建 `gitlink`，那么上述两个命令所代表的含义是不同的。如果 `subdir` 是一个子项目对应的嵌入式 Git 版本库的顶层目录，前者将会创建一个 `gitlink` 引用，同时后者会以 `git add subdir/` 的形式独立添加 `subdir` 目录下的所有文件，那么这也许和用户的预期并不一致。

一种更简单、更好的方案是使用 `git submodule` 命令，它的设计初衷是帮助用户管

理文件内容、元数据、子模块配置信息以及查看和更新它们的状态。为了在子项目中特定目录下将给定版本库添加为子模块，用户可以使用 `git submodule` 命令的子命令 `add`：

```
$ git submodule add https://git.example.com/demo-plugin.git \
  plugins/demo
Cloning into 'plugins/demo'...
done.
```

#### 注意：



在将远程版本库的 URL 替换成路径时，用户需要谨记远程版本库的相对路径和我们的主远程版本库的相对位置，而非我们版本库根目录的位置。

该命令会存储子模块的信息，例如在 `.gitmodules` 文件中的版本库的 URL 地址。如果该文件不存在，系统会自动创建它：

```
[submodule "plugins/demo"]
  path = plugins/demo
  url = https://git.example.com/demo-plugin.git
```

注意，子模块会得到一个和其路径相同的名字。用户可以通过 `--name` 选项显式声明它的名称（或者编辑相关的配置信息）。在子模块目录下执行 `git mv` 命令将会改变子模块的路径，但是会保留其名称。

#### 拉取子模块时的授权验证复用



在存储远程版本库 URL 地址时，一般推荐的做法是存储包含子项目信息的用户名（例如将用户名存放在一个 URL 地址中，例如 `user@git.company.com:mylib.git`）。

不过需要注意的是，在 `.gitmodules` 文件中将用户名作为 URL 地址的一部分来存储是不可取的，因为这个文件必须向其他开发人员公开（通常会使用不同用户名进行授权验证）。幸运的是，子模块中执行的命令能够复用克隆（拉取）自超级项目的验证机制。



如果用户已经添加了一个子模块，并且准备为它添加点感兴趣的内容，`add` 子命令还会为用户执行一个和 `git submodule init` 命令等价的操作。这会把特定子模块配置信息添加到主项目本地配置文件中：

```
[submodule "plugins/demo"]
    url = https://git.example.com/demo-plugin.git
```

上述内容是否似曾相识？为什么 `.gitmodules` 文件和 `.git/config` 文件中的内容是一样的？这是因为 `.gitmodules` 文件的内容是面向所有开发人员的，用户可以根据自身开发环境的实际情况对本地配置文件进行设置。使用两个不同的文件的其他原因是：`.gitmodules` 文件中出现的子模块信息表示只对特定子项目起效，同时出现在 `.git/config` 的相同信息表明它是一个我们感兴趣的给定子模块（我们希望它能够和大家一起共享）。

用户可以通过手动或者执行 `git config -f .gitmodules` 命令创建或者编辑 `.gitmodules` 文件。这一点可能会非常有用，例如用户手动添加和克隆了一个子模块，但是现在又希望使用 `git submodule` 命令操作它。

该文件通常会被提交到超级项目的版本库上（与 `.gitignore`，`.gitattributes` 文件类似），它会被当作子项目出现在文件列表中。



所有其他子命令也需要能够访问这个文件，例如当用户希望给子模块添加内容之前先执行 `git submodule update` 命令，那么会得到类似的下列内容：

```
$ git submodule update
No submodule mapping found in .gitmodules for
path 'plugins/demo'
```

这也是 `git submodule add` 命令会同时暂存 `.gitmodules` 文件和子模块自身的原因：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
new file: .gitmodules
new file: plugins/demo
```

注意，整个子模块其实就是一个目录，就像上述 `git status` 命令查询结果中 `new file` 项展示的那样。默认情况下，大部分 Git 命令都仅限于活动的容器版本库，影响力不会延伸到子模块的嵌套版本库中。如你所见，这是可以配置的。

## 使用子模块克隆超级项目

一个很重要的问题就是，默认情况下，如果用户克隆了超级项目版本库后，将不会得到任何子模块。所有子模块都会从复制的工作目录中消失，只留下它们的基本目录。这一行为是可选择性子模块的基本特性之一。

用户需要告知 Git 系统哪些给定子模块是自己感兴趣的。这可以通过调用 `git submodule init` 命令完成。该命令实际的操作是从 `.gitmodules` 文件中拷贝子模块设置，然后将之复制到超级项目版本库配置中，即 `.git/config` 文件，以便注册子模块：

```
$ git submodule init plugins/demo
Submodule 'plugins/demo' (https://git.example.com/demo-plugin.git)
registered for path 'plugins/demo'
```

`init` 子命令会在 `.git/config` 文件中添加如下内容：

```
[submodule "plugins/demo"]
    url = https://git.example.com/demo-plugin.git
```

用户感兴趣的独立子模块本地配置还支持用户在 `.gitmodules` 文件之外将本地子模块指向不同的 URL 路径（例如某公司对子项目版本库克隆的引用）。

这种机制还可以在子项目版本库被移动时，提供一个新的 URL 地址。这也是本地配置覆盖了 `.gitmodules` 文件配置的原因。否则用户切换到 URL 变更之前的版本时，将无法根据当前 URL 地址拉取变更。另一方面，如果版本库被移动了，`.gitmodules` 文件也进行了相应的更新，那么用户可以通过 `git submodule sync` 命令从本地配置的 `.gitmodules` 文件中重新提取新的 URL 地址。

现在用户已经告知 Git 感兴趣的给定版本库，不过仍然无法从它的远程版本库或者签出到超级项目工作目录下的文件获取子模块的提交记录。用户可以使用 `git submodule update` 命令达到上述目的。



在实际开发工作中，在使用版本库处理子模块时，用户经常会使用 `git submodule update --init` 命令将两个命令组合起来一起执行（`init` 和 `update`）。不过用户至少不需要自定义 URL 地址。

如果用户对所有子模块感兴趣，可以使用 `git clone --recursive` 命令，在克隆操作完成后初始化和更新每个子模块。

为了临时移除一个子模块，并保留后续恢复它的可能性，用户可以使用 `git remote deinit` 命令将之标记为“不感兴趣的（`deinit`）”。这一操作只会影响 `.git/config` 文件。为了永久性地移除一个子模块，用户需要将之标记为“不感兴趣的（`deinit`）”，然后将它们从 `.gitmodules` 文件和工作区（使用 `git rm` 命令）中分别移除。

### 超级项目发生变更后更新子模块

为了更新子模块，使得工作区中的内容能够及时反映子模块在当前版本超级项目中的状态，用户需要执行 `git submodule update` 命令。该命令会更新子项目中的文件，在必要情况下，还会克隆和初始化子模块版本库：

```
$ rm -rf plugins/demo      # clean start for this example
$ git submodule update
Submodule path 'plugins/demo': checked out '5e28a713d8e87...'
```

`git submodule update` 命令会去访问 `.git/config` 文件中引用的版本库，拉取在索引中找到的提交对象的 ID，并将该版本签出到 `.git/config` 文件指定的目录。当然用户也可以声明希望更新的子模块，将子模块路径作为命令执行参数进行传递。

因为这里是根据 `gitlink` 签出给定修订版本，而非根据一个分支，`git submodule update` 命令将会使得子项目的 HEAD 脱离（如图 9-3 所示）。该命令会将子项目直接回退到超级模块中的原始版本。

下面是一些用户需要知道的注意事项。

- 如果用户修改了超级项目当前修订的版本，例如编辑分支、使用 `git pull` 命令导入分支、使用 `git reset` 命令回退历史记录，那么用户需要执行 `git submodule update` 命令将相关的内容更新到子模块。这一操作并不能自动完成，



因为它可能会间接导致用户失去子模块中的工作成果。

- 相反,如果用户切换到其他分支,或者以其他方式修改了超级项目中当前修订版本,并且没有执行 `git submodule update` 命令, Git 系统会认为用户修改了子模块目录而特意指向一个新的提交对象(其实它是一个已有的提交,用户对它进行了编辑,但是忘记更新它的状态)。如果在这种情况下,用户又意外地执行了 `git commit -a` 命令,那么将会修改 `gitlink`,这会导致一个不正确的子项目版本被存储到了超级项目的历史记录中。
- 用户可以在子项目中使用普通的 Git 命令拉取(切换)希望获得的子模块版本,以便更新 `gitlink` 引用,然后在超级模块中提交该版本。这里用户不需要使用 `git submodule` 命令。

用户在从主项目的远程版本库上获得更新的同时,可以让 Git 自动拉取和初始化子模块。该操作可以通过 `fetch.recurseSubmodules` (或者 `submodule.<name>.fetchRecurseSubmodules`) 进行配置。这个配置的默认值可以按需配置(如果是 `gitlink` 的变更,可以是 `fetch`,以及子模块提交指向的引用丢失了)。用户可以通过将之设置为 `yes` 或者 `no` 来启用递归拉取子模块信息,或者无条件禁用该功能。相应的命令行选项是 `--recurse-submodules`。

不过这还不是重点需要关注的,因为 Git 能够自动拉取子模块信息,但是它不能执行自动更新操作。用户本地的子模块版本库可以通过子模块的远程版本库进行及时更新,但是子模块工作目录的状态仍然停留在未更新之前。如果用户没有显式声明更新子模块的工作目录,那么在容器版本库中的下一个提交将会回退该子模块。当前还没有配置选项或者命令行选项可以在系统拉取更新时自动更新所有自动拉取的子模块。不过在不远的将来, Git 系统应该会在管理子模块方面有所改进。

注意,除了在已脱离 HEAD 上签出 `gitlink` 化的修订版本之外,用户还可以在子模块中通过 `--merge` 选项,将超级项目中的提交记录合并到当前分支上;或者使用 `--rebase` 选项对 `gitlink` 之上的当前分支进行变基,就像使用 `git pull` 命令一样。子模块版本库默认会采用 `master` 分支。不过分支名可以通过在 `.gitmodules` 或 `.git/config` 文件中的 `submodule.<name>.branch` 选项进行覆盖,后者的优先级更高一些。

如你所见, `gitlinks` 和 `git submodule` 命令的使用非常复杂。从根本上来说, `gitlink` 的概念可能更适合处理子项目和超级项目之间的关系,但是正确地使用这种信息的难度大大超出了用户的预期。另一方面,它也提供了更强大的功能和灵活性。



## 查看子模块变更

默认情况下，`status`、`log` 和 `diff` 命令的输出结果都是完全基于活动版本库的状态的，并且其影响不会延伸到子模块中。这往往是有问题的，用户需要记得执行 `git submodule summary` 命令进行查看。如果用户受限于此，那么就很容易将一个错误掩盖：用户会发现子模块发生了变更，但是无法知道变更是如何发生的。

不过用户可以通过配置变量 `status.submoduleSummary` 来告知 Git 系统，让它使用能够感知子模块信息的 `status` 命令。如果该变量被设置为一个非零值，那么该数字将会提供 `--summary-limit` 的限制数目；如果该值为 `true` 或者 `-1`，那么代表对数目是无限限制的。

设置该变量之后，用户将会得到类似下列内容的额外信息：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
new file:   .gitmodules
new file:   plugins/demo
```

```
Submodule changes to be committed:
```

```
* plugins/demo 0000000...5e28a71 (3):
  > Fix repository name in a README file
```

`status` 命令扩展将会永久显示子模块变更的相关信息(`new file: plugins/demo`)，`plugins/demo` 文件下子模块的变更信息是新增了 3 个提交，并且会显示最后一个提交的摘要信息（在 README 文化中修正版本库的名称），摘要信息前面向右的尖括号 `>` 表示提交记录已经被添加，即已经在工作目录中出现，但是还未出现在超级项目的提交历史记录中。



实际上，添加的部分只是 `git submodule summary` 命令的输出结果。

对于有问题的子模块来说，在给定超级项目提交中的子模块版本和工作区（以前的版本可以使用 `--cached` 选项显示）或者索引中的子模块版本之间的一系列子模块提交将会被显示。还可以通过 `git submodule status` 命令显示每个模块的摘要信息。

`git diff` 命令的默认输出结果也并不会显示子模块内部变更的详细信息，它只会告诉用户其中存在差异：

```
$ git diff HEAD -- plugins/demo
diff --git a/plugins/demo b/plugins/demo
new file mode 160000
index 0000000..5e28a71
--- /dev/null
+++ b/plugins/demo
@@ -0,0 +1 @@
+Subproject commit 5e28a713d8e875f2cf1060c2580886dec3e5b04c
```

幸运的是，命令行选项 `--submodule=log` 可以告知用户更多有用的信息（用户可以通过 `diff.submodule` 配置变量默认启用该功能）：

```
$ git diff HEAD --submodule=log -- plugins/demo
Submodule subrepo 0000000...5e28a71 (new submodule)
```

除了日志之外，用户还可以使用简短格式只显示提交的名称，如果未指定格式的话，默认的输出结果就是该格式（即只执行 `it diff --submodule` 命令）。

## 从子模块上游获取更新

需要提醒用户的是，子模块提交在 `gitlinks` 中是使用 `SHA-1` 标识符进行引用的，它同时也是为了解决同一版本冲突而存在的；它和分支名这样的不稳定引用是不一样的。因此，超级项目中的子模块并不会自动更新（可能会破坏应用程序的可用性）。但是有时用户也许希望更新它。

假定子项目版本库获得了新发布的修订，接下来用户希望把超级项目中的子模块更新到最新版本。

为了达到这个目的，用户需要更新子模块的本地版本库，移动超级项目工作目录下我

们想要的版本，最后在超级项目中提交子模块的变更。

用户可以手动完成上述步骤，首先把当前工作目录切换到子模块的工作目录下，然后在子模块内部执行 `git fetch` 命令，获取本地版本库克隆的数据（在超级项目的 `.git/modules/` 目录下）。在使用 `git log` 命令验证获取的数据之后，用户就可以更新工作目录了。如果本地版本库中没有新增变更记录，用户就可以简单地签出目标修订版本即可。最后，用户需要在超级项目中创建一个提交。

除了能够更细粒度地控制之外，这个方法的另外一个优点是它不会受到用户当前工作状态的影响（无论用户在一个活动分支，还是一个脱离 HEAD 的匿名分支）。

另外一种解决方案是使用容器版本库进行处理，使用 `git submodule update --remote` 命令，通过远程跟踪分支显式更新子模块。和普通的 `update` 命令类似，用户可以选择合并或者变基来替代分支的签出。

用户还可以使用配置变量 `submodule.<name>.update` 和配置默认上游的配置变量 `submodule.<name>.branch` 来配置默认的更新方式。



总之，`submodule update --remote --merge` 命令会合并上游的子项目变更到子模块，同时 `submodule update --merge` 命令会合并超级项目的 gitlink 变更到子模块。

`git submodule update --remote` 命令将会自动拉取子模块远程版本库主机的新增变更，除非用户指定了 `--no-fetch` 选项。

## 发布子模块变更到上游

在在子模块中直接添加变更时，忘记推送子模块变更到服务端是非常危险的问题之一。一个好的子模块开发习惯是首先提交变更到子模块，推送模块变更，然后返回容器父项目，提交变更，最后推送容器变更。如果用户只推送超级模块版本库而忘记推送子模块，那么其他开发人员在获取更新时将会遇到问题。

虽然 Git 在获取超级项目更新时不会对用户发出警告，但是用户在执行 `git submodule summary` 命令（如果配置正确，还会在 `git status` 命令的输出结果中发现）和尝试更新工作区时就会发现问题所在：

```
$ git submodule summary
```

```
* plugins/demo 12e3a52...0e90143:
Warn: plugins/demo doesn't contain commit 12e3a529698c519b2fab790...
$ git submodule update
fatal: reference is not a tree: 12e3a529698c519b2fab790...
Unable to checkout '12e3a529698c519b2fab790...' in submodule path 'plugins/
demo'
```

如上所述，用户可以清楚地知道记住推送子模块是非常重要的。如果有必要的话，用户可以使用 `git push --recurse-submodules=on-demand` 命令要求 Git 在推送超级项目时自动推送子模块（其他选项只是用来做校验的）。在 Git 2.7.0 或者更高版本的软件中，用户还可以使用配置变量 `push.recurseSubmodules` 达到该目的。

### 9.1.5 将子文件夹迁移到子树或者子模块中

在 Git 中考虑子项目的应用场景时首先需要注意的问题是，项目中是否有相应的源码适合这种划分。子模块和子树都是以超级项目（主项目）的子目录形式存在的。用户无法在一个目录中将分别属于不同子系统的文件混合在一起。

经验表明，大部分系统都使用了这样的目录层次，即使是在大型版本库中也是如此，这是模块化开发的良好开端。因此，将一个子文件夹转换成一个真实的子模块/子树是非常容易的，具体包含以下几个步骤：

（1）将目标子目录移动到超级项目工作区之外，使其和超级项目的顶层目录并列。如果子项目的历史记录非常重要、希望予以保留，可以考虑使用 `git filter-branch --subdirectory-filter` 命令或者其他诸如 `reposurgeon` 这样的工具来清理历史记录。详情可以参考第 8 章。

（2）使用子项目版本库名称重命名该目录，以便更好地表达被提取组件的本质。例如一个名为 `refresh` 的普通子目录可以重命名为 `refresh-client-app-plugin`。

（3）为子项目创建公共版本库（上游），将之作为一个普通的项目（例如在 GitHub 上创建一个新项目来保留被提取的代码，以相同的组织形式将之作为一个超级项目，又或者以特定的组织形式将之作为一个应用程序插件）。

（4）使用 `git init` 命令初始化一个自给自足的标准插件作为 Git 版本库。如果用户已经把第一步提取的子目录历史记录分配到某些分支上，那么就将该分支推送到刚才创建的版本库。将第 3 步创建的公共版本库作为默认的远程版本库，并通过刚创建的 URL 推送初始化提交，以便存储子项目代码。



(5) 在超级项目中, 读取刚才提取的子项目, 并根据实际情况将之作为子模块或者子树, 具体的解决方案可以根据用户的偏好和实际需要进行适当调整。使用刚才创建的子项目公共版本库的 URL 即可。

(6) 在超级项目中提交变更, 然后推送变更到公共版本库, 其中还包括子模块中新增(或者只是经过编辑) `.gitmodules` 文件的情形。

将子目录转换成独立子模块比较推荐的做法是使用一个只读的 URL 进行子模块克隆(添加)。这意味着用户既可以使用 `git://` 协议地址(警告: 这种情况下没有验证授权机制), 也可以使用不包含用户名的 `https://` 协议地址。该做法的目标是通过移动子模块代码到一个独立的子项目版本库来实现隔离。为了确保子模块提交对其他开发人员都是可以访问的, 每个变更都会添加到子项目的公共版本库上。

如果这种做法(最佳实践)完全行不通, 那么在实际开发工作中, 用户可以直接在超级项目的子项目源代码上工作, 不过这样一来就更容易出错。用户需要谨记是, 在嵌套的子模块子目录中, 首先提交和推送子模块, 否则其他开发人员将不能获得变更记录。这种组合方案可能更简单易用, 但是却舍弃了实现和变更处理之间的真正隔离, 因此在使用子模块时更应该深思熟虑。

### 9.1.6 子树和子模块

一般而言, 子树更易用并且也不需要什么技巧。很多人选择子模块, 这是因为其包含了更多 Git 内置工具的支持(例如它有自己的 Git 命令 `git submodule`), 详尽的说明文档, 以及类似 Subversion 的外部工具等, 使得人们一厢情愿地以为它更容易上手。添加一个子模块非常简单(只需运行 `git submodule add` 命令即可), 特别是在没有 `git subtree` 或者 `git stree` 这类第三方工具辅助的情况下创建一个子树。

子树和子模块之间的主要差异在于, 子树只有一个版本库, 这意味着它只有一个生命周期。子模块和类似方案使用嵌套的版本库, 其中每个版本库都有自己的生命线。

虽然子模块容易设置并且灵活性也比较强, 但是操作时也因此容易犯错, 用户需要多加练习、谨慎操作, 才能够完全驾驭它。事实上子模块是有可选择性的, 这也意味着涉及子模块的变更时, 需要每个协作开发者手动更新它。子树也是如此, 因此获取超级项目的变更对于子项目来说在操作上都是一样的。

诸如 `status`、`diff` 和 `log` 这类命令会显示和子模块有关的一些重要信息, 除非进行了跨版本库级别的适当设置, 否则用户很容易错过一个变更记录。对于子树, `status` 命令表现正常, 但是 `diff` 和 `log` 命令需要特别留意, 因为子项目提交的根目录是不同的。后者假定用

户不打算包含子项目的历史记录（通过压缩子树合并）。在子项目版本库中只使用远程跟踪分支时才会出问题。

因为各自独立的不同版本库的生命周期是不一样的，在一个容器项目中更新一个子模块时，需要执行两次提交和两次推送操作。更新一个子树合并的子项目非常简单：只需要一次提交和一次推送操作。

换句话说，使用子模块发布子项目变更到上游更简单，使用子树需要提取变更集（这里 `git subtree` 工具可能会帮上大忙）。

接下来的主要问题，也是根源性的问题，即子模块拥有当前修订的两个源数据：一个是超级项目中的 `gitlink`，另外一个子模块版本库克隆中的分支。这意味着 `git remote update` 命令的工作原理有一点和侧面推动到一个非裸版本库类似（详情可以参考第6章）。子模块的首部（HEAD）一般已经被分离了，因此任意本地更新都需要若干预备操作来避免创建一个无效提交。子树不存在这样的问题。子树中所有修订变更的命令和普通的命令没什么区别，在不需要任何额外操作的情况下，能够给子项目目录添加正确的修订记录。从子项目版本库获取变更记录只需要执行子树合并即可。其和普通的 `pull` 操作唯一的区别在于执行命令时会添加 `-s subtree` 选项。

当然，有时使用子模块是正确的选择。和子树相比，它允许一个子项目（模块）禁用拉取更新，当用户的代码基非常庞大时这会非常有用。在开发堆栈的生态系统中，代码不能被很好地模块化处理时，子模块也会非常有用。

子模块自身也可以是其他子模块的超级项目（容器），以便能够创建一个层级式的子项目。能够更方便地使用子模块嵌套得益于 `git submodule status`、`update`、`foreach` 和 `sync` 等子命令都支持 `--recursive` 选项。

## 子树用例

使用子树时只涉及一个版本库，不会遇到版本库嵌套的情况，这和一个普通的代码仓库类似。这意味着其中只包含一个生命周期。子树的主要特色之一就是能够在一般的修复和功能增强特性中加入特定容器的自定义特性。项目可以通过用户认为最合逻辑的形式进行组织和划分。使用单个版本库还能降低依赖管理的成本。

子树最基本的用法是管理一个程序库的自定义版本以及一个外部依赖。搭建开发环境执行编译和测试也很简单。`Monorepo` 软件甚至可以让所有项目包含统一的版本号的过程可视化。跨子模块之间的原子提交也成为可能。因此单个版本库也能一直保持状态一致。

用户还可以使用子树嵌入相关的详情，例如一个 GUI 工具或者 Web 接口。事实上，很多子模块用例也适用于子树方案，除了备选子项目以及不同项目拥有不同访问权限等特例之外。在这些情况下，用户需要使用子模块。

### 子模块用例

在子模块使用方面争论最激烈的是模块化的问题。本书涉及的子模块使用领域主要是处理插件和扩展。某些编程语言生态系统，例如 ANSI C、C++ 以及 Objective-C，对于管理版本锁定的多模块项目缺乏良好的支持。在这种情况下，可以使用子模块将类插件的代码添加到应用程序中（主项目），继而不必牺牲从版本库将插件更新到最新版本的便捷性。如何拷贝插件的操作指令的常用方案一般会写在 README 文件中，这么做是为了将它和历史元数据隔离。

这种架构还可以扩展到不可编译的代码，例如 Emacs 列表设置、dot 文件的配置（其中包括像 oh-my-zsh 这样的框架工作），以及主题皮肤（也适用于 Web 应用）。在这些情况中，经常需要用到组件的地方是主项目树中规范路径上模块代码的物理位置，这是由技术和采用的框架决定的。例如 Wordpress 和 Magento 的主题、插件等组件实际上都是以这种方式安装的。大部分情况下，用户需要在超级项目中对这些可选组件进行测试。

当然，子模块的另外一个特殊用法是对高级应用基于访问控制和可见性约束的划分。例如项目中可能会使用包含授权限制的加密代码，仅限于一小部分开发人员访问。子模块中的这类代码可以对它的版本库添加访问限制，其他开发人员将不能简单地克隆该子模块。在这种方案中，如果该代码不能访问，编译系统需要能够自动跳过这类加密组件。另一方面，专用的编译服务器可以通过这种方式进行配置，使得客户端获得的应用程序编译支持加密。

一个和可见性限制用途类似、效果却相反的方法是让源码在发布之前，让它的示例支持长期访问。这可以得益于社区共同贡献而获得质量更佳的代码。将主版本库制作成一本自身封闭的书（私有的），但是提供一个包含子目录的 examples/ 目录，在其中引入源代码的示例文件，并让这个子版本库支持公开访问。在生成该书的 PDF 和 EPUB 格式时（或许还有 mobi 格式），编译过程能够像处理普通目录那样将这些示例嵌入书中（或者其中的片段）。

### 子项目第三方管理方案

如果用户发现 git subtree 和 git submodule 命令并不能满足需要，那么可以尝试使用为数众多的第三方项目工具来管理依赖、子项目或者一组版本库集合。这类工具之一是一个



由 Miles Georgie 开发的外部项目。用户可以通过如下网址了解详情：<http://nopugs.com/ext-tutorial>。该项目是与 VCS 无关的，能够用来管理采用了任意版本控制系统组合的子项目和超级项目。

另外一款 repo 工具 (<https://android.googlesource.com/tools/repo/>) 是 Android 开源项目用来统一管理多个版本库跨网络操作的。可供用户选择的类似工具非常多。



当在原生支持功能和众多版本库工具之间做出选择之后，用户应该检查它在类子树和类子模块方面的能力，以便确认它是否能够满足用户的项目要求。

## 9.2 大型 Git 版本库管理

由于 Git 天生的分布式特性，其会在每个版本库拷贝中保留完整的历史记录。每个克隆不仅获得了所有文件，还获得了每个文件每次提交的修订版本。这非常有利于开发（本地操作并不涉及网络传输，响应速度足够快，因此并不会成为影响效率的瓶颈）和团队协作（原生的分布式特性支持多种协作工作流）。

但是当用户需要处理的版本库非常庞大时又该如何呢？用户能够避免耗费大量硬盘空间来应对版本控制存储么？是否可以减少最终用户检索版本库克隆的数据量？

如果用户仔细想过，那么造成版本库极速膨胀的原因主要有两个：用户可能积累了很长的历史记录（每个修订方面），也可能是包含了大量需要和代码一起管理的二进制资源文件，或者二者兼而有之。对于这两种情况，解决的方法和涉及的技术是完全不同的，可以对它们分而治之。

### 9.2.1 处理包含大量历史记录的版本库

虽然 Git 能够高效地处理拥有长历史记录的版本库，比较古老的项目一般拥有的修订数量也非常庞大，这使得与之相关的克隆操作会非常耗时。大部分情况下，用户对于古老的历史记录并不感兴趣，也不希望把时间浪费在获取项目所有修订记录上，更不愿使用额外的硬盘空间存储它们。

例如当用户希望开发一个新功能或者创建一个 bugfix 时（后者可能需要用户在本机执行 `git bisect` 命令，该操作很容易重复产生功能缺失性的 bug，详情可以参考第 2 章），但是



不希望浪费时间等待克隆完整的版本库，因为这一操作可能耗时很久。



某些 Git 版本库托管服务，例如 GitHub 还提供了基于 Web 的接口来管理版本库，其中包括基于浏览器的文件管理和编辑。它们甚至能够自动为用户创建版本库的副本，以使用户可以编辑和提交变更记录。

不过基于 Web 的接口并不是完美无缺的，用户可能还会使用自托管版本库或不提供此功能的服务。

## 使用浅克隆获取截断式历史信息

快速克隆并节省硬盘空间的简便方法是使用 Git 执行浅克隆操作。该操作允许用户获得特定深度截断式的本地版本库副本，其中的深度是指最新修订记录的数量。

那么该如何做呢？只需使用 `--depth` 选项即可：

```
$ git clone --depth=1 https://git.company.com/project
```

上述命令只会克隆主分支最近的若干修订版本。该技巧能够节省大量的时间，并且还能大大减轻服务端的负载压力。一般来说，浅克隆执行完毕的时间是以秒为单位而非以分钟为单位的。这是 Git 比较显著的一个改进。

Git 自 1.9 版以来虽然还有不少问题，但是已经能够支持 `pull` 和 `push` 命令的浅克隆操作了。用户可以通过 `--depth=<n>` 选项修改浅克隆在执行 `git fetch` 命令时的深度（注意这并未获取深度提交对象的标签）。如果想将一个简化的版本库转换成完整版本库，使用 `--unshallow` 选项即可。

注意，`git clone --depth=1` 命令可能仍然会获取所有分支和标签。如果远程版本库没有 `HEAD` 时，这是可能发生的，这是因为它没有主分支可供选择，否则将会只获取上述单个分支的外部引用。长期项目在它的历史记录中一般会包含多个预览版程序。为了切实地节省时间，用户需要和浅克隆搭配使用的是如下方案：分支约束。

## 仅克隆单个分支

默认情况下，Git 会克隆所有分支和标签（如果用户希望获取笔记和置换集，就需要显式声明这一点）。可以通过声明用户希望克隆的单个分支限定历史记录的数量：

```
$ git clone --branch master --single-branch \
https://git.company.com/project
```

因为除了极个别情况外，大部分项目历史记录（基本上是修订的 DAG）都是分支间共享的，用户也许不希望在使用该命令时看到大量的差异记录。

如果用户不希望获得已脱钩的孤儿分支，或者与此相反——用户只希望获得一个孤儿分支（例如项目的 Web 页面），那么该特性将会非常有用。它和浅克隆一起协同工作的兼容性也非常好。

## 9.2.2 处理包含大量二进制文件的版本库

在某些特殊情况下，用户也许需要在代码基中跟踪大量的二进制资源文件。游戏开发团队必须处理大量的 3D 模型，Web 开发团队需要跟踪 raw 图片资源或者 Photoshop 文档，并且上述两种团队可能都需要对视频文件进行版本控制。有时用户也许会发现获得包含二进制交付文件的便利性非常困难，并且代价高昂，例如存储一个虚拟机镜像的快照文件。

Git 在处理二进制资源方面也做了一些微调。因为二进制文件版本之间的变更非常明显（并不只是修改某些元数据首部信息），用户也许会希望在 `.gitattributes` 文件中为某些特定类型的文件显式指定 `-delta` 选项，以便禁用增量压缩（详情可以参考第 4 章和第 10 章）。Git 将会为超过变量 `core.bigFileThreshold` 值的任意文件自动启用增量压缩功能，该变量的默认值为 500MB。用户也许会希望禁用文件压缩功能（例如当一个文件已经是压缩格式出现时）；因为 `core.compression` 和 `core.looseCompression` 变量的影响范围是整个版本库，那么如果二进制资源位于一个独立的版本库（子模块）中，这就是非常有意义的。

### 将二进制程序集文件夹分割到独立子模块中

一种处理大型二进制资源文件夹的可选方案如前文所述，即将它们分割到独立的版本库中，然后将这些资源作为子模块添加到用户的主项目中。采用子模块后，它可以提供一种方法来控制资源文件的更新。不过如果开发者不再需要这些资源文件时，就可以在拉取更新时将资源文件简单地子模块中排除。

该方法的缺点是，用户如果希望以这种方式处理资源文件，那么必须提供一个独立的文件夹来存放这些数量庞大的二进制资源文件。

### 稀疏签出



Git 内置的功能允许用户显式声明经常签出的文件和文件夹。启用该模式是通过将配置变量 `core.sparseCheckout` 的值设为 `true` 实现的，并需要在 `git/info/sparse-checkout` 文件中使用 `gitignore` 语法来声明用户希望在工作目录中出现的文件内容。索引的使用随处可见，可以在签出操作中为文件丢失使用 `skip-worktree` 选项。

如果用户拥有一个大型的树形文件夹，那么这会非常有用，它不会影响本地资源库本身的整体尺寸。

### 在版本库外部存储大型二进制文件

另外一种方案是选用众多第三方工具之一来尝试处理 Git 版本库中的大型二进制文件。它们很多都采用了类似的范式，名义上会将大量二进制文件内容存储在版本库的外部，同时提供某种指针指向签出状态下的内容。每个实现都包含以下 3 个部分：如何存储版本库中受托管文件的信息，如何在团队内部实现大型二进制文件的共享，以及如何与 Git 系统集成（以及存在的性能损失）。在选择方案时，用户需要将上述 3 个问题与实际需求、操作系统的兼容性和易用性，以及研发社区的大小等因素一起通盘考虑。

版本库中存储的是什么，被签入的内容也许是一个文件的系统链接或者密钥，也可能是一个指针文件（通常是纯文本形式），哪些行为会引用实际的文件内容（通过名称或者文件内容的加密哈希函数）。被跟踪文件需要以某种形式存储在后端以方便协作开发（例如云服务、rsync、共享目录等）。后端服务可能需要支持客户端的直接访问，或者提供一个独立的服务器，提供权威的 API 接口来决定哪些 blob 对象可以被写入，哪些需要禁用在外部的加载存储访问。

该工具可能既需要使用独立的命令来签出和提交大型文件，以及拉取和推送到后端服务器，或者还需要和 Git 进行集成。集成方案会使用清洁/涂抹过滤器来处理签出和签入之间的状态转换，以及预推送钩子来自动完成发送大型文件内容的操作。用户只需要声明哪些文件是需要被跟踪的即可，当然还会用到初始化版本库的工具。

基于过滤器的方法的优点是简单易用，不过它存在性能损失，这是由它的工作机制决定的。



使用独立的命令来处理大型二进制资源的学习曲线更陡峭一些，但是它提供了更好的执行性能。某些工具二者兼而有之。

对于不同的解决方案来说，`git annex` 命令背后的开发社区人才济济，并且支持多种后端。同时 `Git LFS`（大文件存储）是由 `GitHub` 推出的服务，它提供了对 `MS Windows` 的良好兼容性、客户端-服务端方法，以及透明性（兼容基于过滤器的方法）。这类工具还有很多，例如 `git-fat`、`git-media`、`git-bigstore` 和 `git-sym` 等。

## 9.3 小结

本章为用户介绍了所有使用 `Git` 管理多组件项目的必备工具，从程序库、图形接口到插件、主题和框架等。

读者学习了子树技术背后的机制，以及如何使用它管理子项目。读者也了解了如何使用子树创建、更新、浏览和管理子项目。

读者同时还学习了可选依赖的嵌套版本库的子模块方法，然后了解了 `gitlinks`、`.gitmodules` 和 `.git` 文件的基本原理。用户在使用子模块时需要特别谨慎，以防因为粗心而导致的陷阱。读者已经对上述故障和陷阱的发生机理了然于心，同时也学习了如何使用子模块创建、更新、浏览和管理子项目。

接下来读者学习了使用子树和子模块的最佳时机以及它们的优缺点，还了解了每种技术的若干用例。

现在用户已经知道如何在一个复杂的环境中高效使用 `Git` 系统，以及有助于用户深入了解 `Git` 系统行为背后的高级理念。第 10 章将会带领读者学习如何更方便地使用 `Git` 系统。



## 第 10 章

# Git 的定制和扩展

前面的章节主要的目标是帮助读者了解和掌握作为版本控制系统的 Git 有关的所有知识，从历史浏览、管理变更记录到团队协作开发，最后以第 9 章处理高级项目结束。

接下来的两章将会帮助用户搭建和配置 Git 运行环境，以便用户（本章）和其他开发人员（下一章）能够高效地进行软件开发。

本章的主旨是通过对 Git 的配置和扩展来满足用户的个性化需求。首先，本章将会向读者演示如何配置 Git 命令行环境，以使它更方便易用。对于某些任务使用可视化工具可能会更简单一些。本章图形化接口的概要介绍将会助用户一臂之力。接下来，本章将会介绍如何配置和调整 Git 的行为，其涵盖范围从配置文件（通过前文所述的声明特定配置选项）到使用 `gitattributes` 文件的单个文件配置。

然后本章将会介绍通过钩子实现某些 Git 工作任务的自动化，相关示例是演示 Git 检查某个项目的提交代码是否符合编码规范。这部分的重点是客户端钩子，简要介绍的服务端钩子将会在第 11 章详细阐述。本章最后一部分内容将会介绍如何扩展 Git，其范围包括命令行别名、集成新的用户可视化命令、助手程序和驱动（新的后端兼容性）等。

其中很多知识点，例如 `gitattributes`、远程助手、凭据助手以及 Git 的基本配置，在前面章节已经有所涉猎。本章将会集中介绍这些内容，并进一步对它们进行深入扩展。

本章的主要内容包括以下几个方面。

- 为命令行配置 shell 提示符和 Tab 键自动补全功能。
- 图形化用户接口的类型和示例。
- 配置文件和基本的配置选项。
- 多种钩子的安装和使用。

- 简单和复杂的别名应用。
- 使用新命令和助手程序扩展 Git。

## 10.1 Git 与命令行

使用 Git 版本控制系统的方式有多种。其中包括大量支持多种场景和功能的图像化用户接口，还有不少工具和插件支持集成 Git 到某些集成开发环境或者文件管理器中。

不过命令行是唯一可以执行所有 Git 命令和与之相关的命令行选项的地方，如果用户希望尝试 Git 的某些新特性，官方首先会以命令行的形式提供。当然，大部分 GUI 工具只实现了 Git 所有功能的某个子集。掌握命令行的使用也能确保用户深入理解上述工具、机制及其功能。只知道 GUI 的使用对于掌握 Git 基础知识来说是远远不够的。

无论用户使用 Git 命令行是基于自身选择，又或者是环境偏好，还是因为它是执行用户所需功能的唯一访问方式，它提供的不少 shell 特性都能够使得 Git 的使用体验更人性化一些。

### 10.1.1 Git 命令行提示符

自定义用户自己的 shell 提示符，让它能够显示用户当前所在版本库的状态将会非常有用。



shell 命令行提示符是一些短信息，它可以被写入终端或者显示在控制台输出结果中，以便响应交互式 shell 的用户输入（通常是一个 shell 命令）。

这类信息丰俭由人。Git 提示符也可以和普通的命令行提示符类似，或者外观上存在显著差异（当用户在 Git 版本库内部时很容易区分它们）。

在 contrib/区域中有一个 bash 和 zsh shell 的样本实现。如果用户是从源码安装 Git 的，只需拷贝 contrib/completion/git-prompt.sh 文件到自己的主目录即可。如果用户是在 Linux 上通过包管理器安装 Git 的，那么它的文件位置可能是/etc/bash\_completion.d/git-prompt.sh。该文件提供的 \_\_git\_ps1 函数可以在 Git 版本库中生成一个 Git 命令行提示符，不过用户首先需要将它们作为源文件添加到 .bashrc 或者 .zshrc 文件中：

```
if [ -f /etc/bash_completion.d/git-prompt.sh ]; then
    source /etc/bash_completion.d/git-prompt.sh
fi
```

shell 提示符可以通过配置调用环境变量。为了配置提示符，用户必须直接或者间接地修改 PS1 环境变量（提示符字符串之一，默认的交互提示符）。为此，一种办法是通过命令替换机制在 PS1 环境变量中创建一个包含调用 `__git_ps1` shell 函数的 Git 命令提示符：

```
export PS1='\u@\h:\w$( git_ps1 " (%s)" )\$ '
```

注意，对于 zsh，用户还需要在 shell 中使用设置选项命令 `setopt PROMPT_SUBST` 启用命令替换机制。

此外，为了更轻便、快捷地使用提示符，并让它支持彩色主题外观，用户可以使用 `__git_ps1` 来设置 PS1。为此，用户可以在 bash 中通过设置环境变量 `PROMPT_COMMAND`，在 zsh 中通过调用 `precmd()` 函数。用户可以在 `git-prompt.sh` 中的注释信息中找到更多与此选项有关的信息。对于 bash，它可能和如下内容类似：

```
PROMPT_COMMAND=' git_ps1 "\u@\h:\w""\\$ " " (%s)"'
```

使用该配置（以及相关的解决方案）之后，命令提示符的外观如下所示：

```
bob@host.company.org:~/random/src (master)$
```

bash 和 zsh shell 提示符可以通过能够扩展 shell 的特殊字符进行自定义。在本示例中，`\u` 代表当前的用户（bob），`\h` 代表当前的主机名（host.company.org），`\w` 代表当前的工作目录（~/randomm/src），同时 `\$` 会打印 `$` 部分的提示符（假定用户是以 root 用户登录的）。PS1 中的 `$(...)` 是用来调用外部目录和 shell 函数的。这里的 `__git_ps1 " (%s)"` 是用来调用 `git-prompt.sh` 提供的 shell 函数 `__git_ps1` 的。其中还包括一个格式化参数：`%s` 是用于表示 Git 状态的占位符。注意，用户既可以在命令行中设置变量 PS1 时使用单引号，如本示例所示；也可以对 shell 替换方法进行转义，以便在显示提示符时能够进行正确的替换，而非在定义该变量时。

如果用户正在调用 `__git_ps1` 函数，Git 还会显示当前正在进行的多步操作的信息：合并、变基、二分查找等。

例如在 master 分支上执行交互式变基时（-i），和提示符相关的部分将会是 `master|REBASE-i`。这里的命令行中显示这类信息将会非常有用，特别是用户在操作过



程中因故中断时。

命令行中还可以显示工作树、索引等区域的状态。如表 10-1 所示，用户可以通过导出这些环境变量的选定子集来启用这些特性（对于某些特性，用户可以根据单个版本库的布尔值配置变量对它们进行禁用）。

表 10-1 环境变量的属性及作用

配置变量	值	作用
GIT_PS1_SHOWDIRTYSTATE bash.showDirtyState	非空	*表示未暂存的变更，+表示已暂存变更
GIT_PS1_SHOWSTASHSTATE	非空	如果某些内容被暂存了，将会以\$符号显示 is stashed.
GIT_PS1_SHOWUNTRACKEDFILES bash.showUntrackedFiles	非空	如果工作目录中包含未跟踪文件，将会以%符号标识
GIT_PS1_SHOWUPSTREAM bash.showUpstream	空格分隔的列表值： <ul style="list-style-type: none"> <li>• verbose</li> <li>• name</li> <li>• legacy</li> <li>• git</li> <li>• svn</li> </ul>	这会自动显示用户是否在 upstream.name 之后 (<)，当时 (=) 和之前 (>) 显示上游名称；verbose 表示之前/之后的提交数量（使用一个签名）；git 表示会比较 HEAD 和 @{upstream}；svn 表示会和 SVN 上游进行比较
GIT_PS1_DESCRIBE_STYLE	备选参数值列表： <ul style="list-style-type: none"> <li>• contains</li> <li>• branch</li> <li>• describe</li> <li>• default</li> </ul>	这在脱离 HEAD 时提供了额外的信息。contains 表示使用更新的附注标签。branch 表示使用更新的标签或者分支，describe 表示使用较旧的附注标签，default 表示如果存在匹配的标签，将会予以显示
GIT_PS1_SHOWCOLORHINTS (仅限提示符/precmd)	非空	用彩色表示当前目录的非整洁状态
GIT_PS1_HIDE_IF_PWD_IGNORED bash.hideIfPwdIgnored	非空	如果当前的目录根据设置被 Git 系统忽略，那么将不会显示 Git 命令提示符



如果用户使用的是 `zsh shell`，那么可以查看 `zsh-git` 脚本集、`zshkit` 配置脚本或者 `zsh` 的 `oh-my-zsh` 框架，以便替代使用 `Git` 的 `contrib/`区域自动补全和命令行提示符配置脚本 `bash-first`。此外，用户还可以使用 `zsh` 内置的 `vcs_info` 子系统。

也有 `bash` 的替代性命令行提示符解决方案可供用户选择，例如 `git-radar`。



当然，用户也可以生成自己的 `Git` 命令提示符工具，例如用户也许会希望在 `git rev-parse` 命令的帮助下，将当前的工作目录分割为版本库路径部分和项目子目录路径部分。

### 10.1.2 Git 命令自动补全

另外一个方便 `Git` 命令行操作的 `shell` 特性是可编程的命令行补全功能。该特性可以大大加快用户输入 `Git` 命令的速度。命令行自动补全功能支持用户输入一个命令或者文件名的头几个字符，然后按下自动补全键（通常是 `Tab` 键）补全其余的字符。通过 `Git` 命令自动补全功能，用户还可以实现子命令、命令行参参数、远程版本库名称、分支名和标签的自动补全，并且只在适当的情况下进行补全操作（例如只对命令行中预定位置的远程版本库名称进行自动补全）。

`Git` 为 `bash` 和 `zsh` 等 `shell` 内置了（不过是可选安装的）`Git` 命令的自动补全支持。

对于 `bash`，如果自动补全功能没有和 `Git` 一起安装（在 `Linux` 系统下，默认位置是 `/etc/bash_completion.d/git.sh`），那么用户需要从 `Git` 源码中将源文件 `contrib/completion/git-completion.bash` 拷贝到某个目录，例如用户自己的主目录下，然后在用户的 `.bashrc` 或者 `.bash_profile` 文件上对它进行来源标记：

```
.. ~/git-completion.bash
```

`Git` 的自动补全功能一经启用，用户就可以在命令行中对它们进行输入测试：

```
$ git check<TAB>
```

启用自动补全功能的 `bash`（或者 `zsh`）将会自动补全命令 `git checkout`。

同样，在存在语义模糊的情况下，双击 `Tab` 键会显示所有可能的结果（不过这一特性并非所有 `shell` 都支持）：

```
$ git che<TAB><TAB>
checkout      cherry      cherry-pick
```

自动补全功能还支持命令参数选项的输入，如果用户无法完全记得选项名称而只记得部分前缀，那这将是非常有用的：

```
$ git config --<TAB><TAB>
--add                --get-regex          --remove-section      --unset
--file=              --global              --rename-section      --unset-all
--get                 --list                --replace-all
--get-all            --local               --system
```

除了显示可能性补全候选列表之外，当存在多种可能性补全列表时，某些 shell 还支持转盘式补全列表显示，每按下一一次 Tab 键后，显示一种相同前缀的候选补全列表（循环显示某一类内容）。

注意，命令行自动补全功能只在交互模式下有效，它是基于含义模糊的前缀，而非含义不明的缩写。

### 10.1.3 Git 命令自动校正

Git 内置工具中，和 Tab 自动补全类似但是无任何关联的是自动纠错功能。默认情况下，如果用户疑似因拼写错误输入了一个错误命令，则 Git 会尝试猜测用户的真实意图。即使只有一个候选项，它仍然会拒绝执行该命令：

```
$ git chekout
git: 'chekout' is not a git command. See 'git --help'.
```

```
Did you mean this?
  checkout
```

不过，如果配置变量 `help.autoCorrect` 被设置为一个正数值的话，那么 Git 将会自动纠正输错的命令，并等待给定数值的秒数之后执行该命令（默认 0.1 秒）。如果希望马上执行相关命令，用户可以将该变量的值设置为一个负数值，或者也可以设置为默认值 0：

```
$ git chekout
WARNING: You called a Git command named 'chekout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

```
Your branch is up-to-date with 'origin/master'.
```

系统如果发现输入的文本中包含一个以上的命令，那么将不会执行它们。这种机制是 Git 系统独有的，用户无法自动纠正子命令、参数和选项（和自动补全正好相反）。

### 10.1.4 命令行美化

Git 完全支持彩色的终端输出结果，这大大有利于从视觉上识别命令的输出。有不少选项可以帮助用户选择喜欢的颜色。

首先，用户可以声明相关命令何时使用彩色的输出结果。`color.ui` 是切换颜色的主开关，如果希望禁用所有 Git 的彩色终端输出结果，将它的值设置为 `false` 即可。该配置变量的默认值为 `auto`，这使得 Git 在将执行结果直接输出到终端时会使用彩色表示它们，而当输出结果被重定向到一个文件或者管道时则会忽略颜色控制代码。

用户还可以将 `color.ui` 的值设置为 `always`，尽管用户很少希望这么做。如果用户希望在重定向输出结果中应用颜色代码，那么在 Git 命令中附加 `--color` 选项即可，`--no-color` 选项将会禁用彩色表示的输出结果。

如果用户想进一步对命令和输出结果的颜色进行设置，Git 为用户提供了相应的颜色设置属性：`color.branch`、`color.diff`、`color.interactive` 和 `color.status` 等。和主开关 `color.ui` 类似，上述每种设置都有四种参数值可供选择，它们分别是 `true`、`false`、`auto` 和 `always`。

此外，每种设置还包含一些子设置，用户可以通过它们设定输出结果中特定部分的颜色。这类配置变量的颜色值，例如 `color.diff.meta`（为了在 `diff` 输出结果中配置元数据新的颜色），其中的值是由空格分隔的前景色、背景色（如果存在相关设置）和文本属性一起组成的。

可供用户选择的字体颜色包括以下几种：`normal`、`black`、`red`、`green`、`yellow`、`blue`、`magenta`、`cyan` 和 `white`。对于字体属性，可供用户选择的值包括：`bold`、`dim`、`ul`（下划线）、`blink` 和 `reverse`（使用背景色替换前景色）。

`git log` 命令的美化版输出结果也可以通过相关选项进行颜色设置，详情可以参考 `git log` 官方说明文档。

### 10.1.5 命令行工具替代方案

想知道 Git 的某些用户接口粗糙不堪的原因，只需要记住 Git 是以自下而上的范式进行开发的即可。从历史上看，Git 开始是作为一款开发版本控制系统的工具而存在的（用户可



以在开发文档中找到 Git 早期应用核心指南：<https://www.kernel.org/pub/software/scm/git/docs/gitcore-tutorial.html> 或者 <https://git-scm.com/docs/gitcore-tutorial>）。

第一款 Git “高层命令”的替代品是 Cogito（替代用户接口）。目前 Cogito 已经完成了它的历史使命。它包含的所有特性早就已经集成到了 Git 软件中（或者其他更好的同类产品）。有不少为了达到易学易用的目的而尝试编写的包装器脚本（替代用户界面），例如 Easy Git（eg）。

还有不打算替换整个用户界面的外部 Git 高层命令应用，它们只是提供了某些额外的特性，或者将 Git 封装起来，提供一组有限的功能子集。补丁管理接口，例如 StGit、TopGit 和 Guilt（以前叫 Git Queues（gq）），它们的用途是方便用户重写、编辑和清理某些未发布的历史记录。它们作为交互式变基工具已经在第 8 章相关内容有所提及。另外，还有将 Git 作为后端服务的单文件版本控制系统，例如 Zit 和 SRC。



除了替代性的用户接口之外，Git 还有多种实现（主要用于读写 Git 版本库）。它们的功能完整性各不相同。除了核心的 C 语言实现，还有用 Java 编写的 JGit，以及 libgit2 项目——它是目前多种编程语言的 Git 绑定基础。

## 10.2 图形化接口

现在已经学习了如何通过命令行使用 Git。上一小节向读者介绍了如何对它进行配置和个性化，使得它的运转更高效。不过命令行终端并不是终点，还有其他管理 Git 版本库的方式供用户选择。有时，一套可视化方案可能正是用户梦寐以求的。

接下来将会简要介绍一些以用户为中心的 Git 图形化工具。Git 系统管理工具简介将会放在第 11 章。

### 10.2.1 图形化工具种类

不同的工具和接口都是专门为特定的工作流而开发的。有些工具只支持 Git 功能的某个子集，或者鼓励用户以某种特殊的方式进行版本控制。

为了能够在 Git 图形化工具中做出明智的选择，用户需要了解各种工具都支持的操作类型有哪些。注意，一种工具可能会支持多种类型的操作。



首先要介绍的是图形化历史查看器。用户可以将它看作是 `git log` 命令的强力 GUI 版本。该工具主要是用来帮助用户以可视化方式查看项目过往历史记录和分支布局的。这类工具通常可以接受修订查询相关的命令行选项参数，例如 `--all` 选项。Git 命令行中有 `git log --graph` 命令，以及很少用到的 `git show-branch` 命令，是通过 ASCII 字符码表示历史记录的。

类似的还有图形化的 `blame` 工具，它会显示文件内部每一行的历史记录。对于每一行来说，它会显示该行是何时创建的，以及是何时被移动或者拷贝到当前位置的。用户可以查看每个提交记录的细节，并能够浏览文件中每一行的历史记录。其他类似的应用中，名义上可以查看线性区间演化（`git log -L`）和 `pickaxe` 搜索（`git log -S`）的都没有多少 GUI 可供用户选择。

接下来是主要用于编辑提交记录的提交工具，当然它们通常还包括某些工作树（例如忽略文件和分支切换）管理和远程版本库管理。这类工具通常还可以显示未暂存和已暂存的变更记录，以使用户可以在这些状态之间移动文件。其中某些工具还可以暂存独立的变更代码块，和交互式版本的 `git add`、`git reset` 等命令类似。图形化版本的交互式 `add` 操作在第4章已经详细介绍过，第3章也有所提及。

然后是文件集成管理工具（或者图形化集成 `shell`）。这些插件通常会通过半透明图标显示文件在 Git 中的状态（已跟踪/未跟踪/忽略）。它们还会为文件、目录和版本库提供一个上下文菜单，并辅以键盘快捷键。它们也可能会支持鼠标拖拽操作。

程序员编辑器和集成开发环境（IDE）通常也提供了集成 Git 的支持（或者其他版本控制系统）。这种版本库管理机制（作为团队项目开发的一部分），使得在 IDE 中直接执行 Git 的相关操作，显示当前文件和版本库的状态成为了可能。用户甚至还可以使用版本控制信息给文件视图添加注释。它们通常包含提交工具、远程版本库管理、历史记录查看器和 `diff` 查看器。

Git 版本库的托管服务网站还提供了面向工作流的桌面客户端。它们大部分会将精力集中在一组能够在工作流中辅助协作开发的通用特性上。它们还会自动化常见的 Git 任务，这么做的目的是强调它们可以特供额外特性和集成能力的服务，不过它们可以随时随地与任意版本库托管服务交互。

## 10.2.2 图形化的 `diff` 和 `merge` 工具

图形化差异比较（`diff`）和合并（`merge`）工具的情况有些特殊。对于这类工具，Git 包含集成相关命令的第三方图形化工具，它们分别是 `git difftool` 和 `git mergetool`。这些工具

可以在 Git 版本库中直接调用。需要注意的是，这和外部的 diff 和 merge 驱动是不同的，这些驱动只是简单地替代 git diff 命令，或者增强了相关功能。

虽然 Git 内部有一个 diff 差异比较的实现，以及相应的解决合并冲突机制（详情可以参考第 7 章），但用户仍然可以使用一个外部 diff 工具取而代之。这些外部工具在差异比较方面往往表现更佳（通常是一个包含摘要信息的并行差异比较结果），并且能够辅助用户解决合并冲突（一般会使用三路合并接口）。

配置图形化 diff 工具或者图形化 merge 工具时，有若干自定义属性可供用户设置。为了告知 Git 系统如何选择 diff 和 merge 工具，用户可以分别对 diff.tool 和 merge.tool 进行设置。如果没有设置 “merge.tool” 属性，那么在执行 “git mergetool” 命令时，系统会告知用户如何配置相关信息，并尝试执行某个预定义的工具：

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
tortoisemerge emerge vimdiff
No files need merging
```

执行 git mergetool --tool-help 命令后，将会显示所有可用的工具，其中包括未安装的部分。为了应对希望使用的工具安装路径不在 \$PATH 中或者使用的软件版本不匹配的情况，用户可以使用 mergetool.<tool>.path 来设置或者重写给定工具的路径：

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
vimdiff
[...]
```

```
The following tools are valid, but not currently available:
araxis
[...]
```

```
Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

如果系统没有为用户希望使用的工具提供内置支持，用户仍然可以使用它，只需配置它既可。配置变量 `mergetool.<tool>.cmd` 可以声明如何执行该命令，同时配置变量 `mergetool.<tool>.trustExitCode` 可以告知 Git 相关程序成功执行一个合并操作之后是否退出。配置文件的相应片段如下所示（以名为 `extMerge` 的图形化合并工具为例）：

```
[merge]
    tool = extMerge
[mergetool "extMerge"]
    cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
```

### 10.2.3 图形化接口示例

本小节将会向读者介绍一系列和 Git 有关的工具，或者可以为用户深入了解这些工具起到抛砖引玉的作用。列举出这类软件的 GUI 客户端是一个不错的入门方法。

有两种可视化工具是作为 Git 的一部分和它一起安装的，其名为 `gitk` 和 `git-gui`。它们是用 Tcl/Tk 编写的。`Gitk` 是一款历史记录查看器，而 `git-gui` 是一款提交工具。当然还有 `git gui blame`，它是一款可视化线性历史记录浏览器。这些工具之间是互相联系的，例如可以在 `git gui` 中打开 `gitk` 浏览历史记录。

可视化工具不一定需要用到图形化环境。`tig` 是 Git 基于文本模式的接口，它的功能就是一个版本库浏览器和提交工具，可以看作是 Git 的呼叫器。

还有使用 Python 开发的 `git cola`，它可以兼容所有操作系统，其中包含提交工具、远程版本库管理工具和 `diff` 查看器。

专用于 GNOME 的 `Gitg`，简单并且外观靓丽，它包含图形化的历史记录查看器、`diff` 查看器和文件浏览器。

MacOS 下最流行的开源 GUI 是 `GitX`。该软件有很多衍生版本，`Gitbox` 是比较有趣的版本之一。它的特性包括历史记录查看器和提交工具。

对于 MS Windows，相关的常用工具是 `TortoiseGit` 和 `git-cheetah`，它们都提供了 Windows 上下文菜单集成的支持，因此用户可以在 Windows 资源管理器（文件管理和 shell 接口）中执行 Git 命令。

GitHub 和 Atlassian 这两家公司都发布了桌面 GUI 工具，因此用户可以方便地使用自己的 GitHub 或者 Bitbucket 版本库，而且其功能不仅限于此。除了其他常用的基础功能之外，GitHub 客户端和 SourceTree 在管理版本库方面各具特色。



## 10.3 配置 Git

到目前为止，我们在阐述 Git 工作原理和具体应用的同时，也介绍了若干种改变其行为的方法。接下来将会比较系统地解释如何在临时或者永久性的基础上配置 Git 的操作。同时还会通过引入和重新引入若干重要的配置对 Git 的行为进行自定义配置。通过这些工具，用户使用 Git 时会更加如鱼得水。

### 10.3.1 命令行选项和环境变量

Git 是通过层级方式处理行为变更切换的，从最简单到具体，根据相关的优先级选择最具体的设置（以及最短期限）。

最具体的设置是命令行选项，它会覆盖其他所有设置。受它们影响最明显的是当前的正在执行的 Git 命令。



一个需要注意的问题是，某些命令行选项，例如 `--no-pager` 或者 `--no-replace-objects` 是 git 的包装器，而非 Git 命令本身。例如，通过如下命令可以了解它们之间的区别：

```
$ git --no-replace-objects log -5 --oneline  
--graph  
--decorate
```

用户可以通过如下网址了解 Git 命令行接口使用规范：<https://www.kernel.org/pub/software/scm/git/docs/gitcli.html>。

第二种改变 Git 命令行为的方法是通过环境变量。它们是和当前使用的特定 shell 有关的，如果使用了替换机制，用户还需要通过变量输出将之传递到子过程中。有一些环境变量会影响所有核心的 Git 命令，而有一些只会影响特定的（子）命令。

Git 也充分利用了某些非具体的环境变量。这意味着系统将它们当作最后的手段，它们可以被 Git 中具体的等价物覆盖，例如 PAGER 和 EDITOR 这类变量。

### 10.3.2 Git 配置文件

自定义 Git 工作方式的最后一种方法是使用配置文件。在大部分情况下，配置一



个动作可以通过命令行选项、环境变量和配置变量来完成，上述配置的优先级是依次递减的。

Git 采用了一系列的配置文件来执行可能符合用户意愿的非默认行为。Git 有 3 层钩子用于查找配置变量的值。Git 会从最简单的到最具体的顺序依次读取这些配置文件。配置中后面的属性会覆盖前面的类似属性。用户可以使用 `git config` 访问 Git 的配置信息：默认情况下，它操作的是所有配置文件的联合体，不过用户可以通过命令行选项声明希望访问的特定配置信息。用户还可以使用 `--file=<pathname>` 这样的配置文件语法（例如第 9 章介绍的 `.gitmodules` 文件）访问任何给定的文件（或者使用环境变量 `GIT_CONFIG`）。



用户还可以从任何类似配置文件中的 blob 对象读取数据，例如用户可以使用 `git config--blob=master:.gitmodules` 命令读取 `master` 分支中 `.gitmodules` 文件的信息。

Git 查询配置的第一个位置是系统级的配置文件。如果 Git 是采用默认设置进行安装的，那么该文件应该可以在 `/etc/gitconfig` 目录下找到。至少在 Linux 系统上，文件层级规范（FHS）表明 `/etc` 是存放特定系统范围配置文件的地方。在 Windows 环境下，Git 会将配置文件存放在该程序安装目录下的子文件夹中。该文件中包含当前系统中所有用户和其他版本库相关的属性配置。为了在执行 `git config` 命令时读写特定配置文件（并且使用 `--edit` 选项进行编辑），可以在执行 `git config` 命令时搭配 `--system` 选项。

用户可以使用环境变量 `GIT_CONFIG_NOSYSTEM` 让系统跳过读取这些配置文件。这可以帮助用户建立一个可预见的环境，或者避免使用有问题的配置文件而导致用户难于修复该问题。

Git 接下来要查找的位置是 `~/.gitconfig` 和 `~/.config/git/config`（使用默认配置）。该文件是专属于每个用户的，并且会影响所有用户版本库。如果用户在执行 `git config` 命令时搭配了 `--global` 选项，那么系统会专门从该文件读写信息。注意：在其他地方时，`~`（波浪符号）表示当前用户的主目录（`$HOME`）。

最后，Git 将会查找用户当前使用的 Git 版本库中单个版本库的配置文件信息，即 `.git/config`（默认情况下是非裸版本库）。其中设定的参数值只会影响本地的单个版本库。

用户可以通过`--local`选项要求 Git 系统读写该文件。

每种级别的属性值（系统、全局和本地）都会被上一级的覆盖，例如`.git/config`中的值会覆盖`~/.gitconfig`文件中的值，除非配置变量中包含多个属性值。



用户还可以在单个用户文件中拥有属于自己的默认标识符，并且在必要情况下，在独立版本库中可以使用单个版本库配置文件对它进行覆盖。

## Git 配置文件语法

Git 配置文件是纯文本的，因此用户还可以通过对选定文件手工编辑来自定义 Git 的行为。它的语法灵活宽松，一般会忽略其中的空格（相对于`.gitattributes`来说），井号`#`和分号开头的行到行尾的内容表示注释，空白行会被忽略。

文件内容是由段落和变量构成的，它的语法和 INI 文件的语法类似。段落名和变量名都是大小写敏感的。一个段落是以`[段落名]`的形式开始的，然后一直延续到下一个段落。每个变量必须从某个段落开始，这意味着在设置首个变量之前必须包含一个段落首部。段落可以重复并且允许为空。

段落可以进一步划分为子段落。子段落名也是大小写敏感的，并且可以包含除了换行符之外的任意字符（双引号`"`和斜杠`\`必须分别以`\`和`\\`的形式进行转义）。子段落的起始部分和下列内容类似：

```
[section "subsection"]
```

所有其他行都被当作以 `name = value` 形式的变量设置（注意，其他行是指段落首部之后的行）。有一个特例，那就是 `name` 是 `name = true`（布尔值变量）的简写。这类属性行可以在行尾加上`"\`符号与下一行拼接，即该符号是换行符的转义。行首和行尾空格将会被忽略。属性值内部的空格将会予以保留。用户可以使用双引号保留属性值中首尾的空格。

用户还可以通过设置特定的变量 `include.path` 指定被包含的文件名路径，来达到在某个文件中包含其他配置文件的目。被包含文件将会马上被解析，它和 C 和 C++ 中 `#include` 的机制类似。该路径是相对于配置文件的 `include` 指令的。用户还可以使用 `--no-includes` 选项禁用该功能。

### 配置变量类型和类型声明符



在读取（写入）一个配置变量时，用户可能还需要提供一个类型声明符。其中 `--bool` 选项表示能够确保执行结果的返回值是 `true` 或者 `false`；`--int` 选项表示可选的后缀有 `k` (1024 元素)、`m` (1024k) 和 `g` (1024m)；`--path` 选项表示主目录下 `$HOME` 的值；`~user` 表示主目录中的给定用户。

还有 `--bool-or-int` 选项以及若干与颜色存储和检索有关的转义代码可供用户选择，详情可以参考 `git config` 说明文档。

## 访问 Git 配置

用户可以使用 `git config` 命令访问 Git 配置，从以规范形式查看配置实体列表、单个变量到编辑和添加实体等。

用户可以使用 `git config --list` 命令查询现有的配置，而且可以在必要的情况下添加相应的参数对单个配置层进行约束。在一个包含默认安装的 Linux 镜像中，用户可以在刚创建的 Git 空版本库下执行 `git init` 命令，本地版本库对应的配置如下所示：

```
$ git config --list -local
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
```

用户还可以使用 `git config` 命令查询单个变量键值，将搜索范围限定在特定文件内部或者外部，将给定名称的配置变量作为查询参数（将该参数放置于 `--get` 选项之前），其中包括段落、可选子段落以及由点分隔的变量名（键）：

```
$ git config user.email
alice@example.com
```

上述命令将会返回查询结果的最后一个值，即优先级最高的那个。用户可以使用 `--get-all` 选项获取所有搜索记录，或者使用 `--get-regexp=<match>` 获得指定键值。这在访问远程版本库的 `refspecs` 这样的多值参数时会非常有用。



通过使用 `--get`、`--get-all` 和 `--get-regex` 选项，用户还可以限制查询列表（并设置多值列表），让它只显示符合正则表达式条件的结果（将之作为可选的末尾参数）。例如下列命令：

```
$ git config --get core.gitproxy 'for kernel\.org$'
```

用户可以使用 `git config` 命令设置配置变量的值。例如，为了设置用户的电子邮件名称，该地址一般对于用户所属版本库来说都是通用的，可以执行如下命令：

```
$ git config --global user.name "Alice Developer"
```

为了修改多个变量，用户可以使用如下命令：

```
$ git config core.gitproxy '"ssh" for kernel.org' 'for kernel\.org$'
```

本地层（单个版本库配置文件）默认情况下是支持写入的，除非不需要声明任何参数。对于多值配置选项，用户可以使用 `--add` 选项为它添加多行配置，还可以使用 `git config --unset` 命令方便地删除配置实体。

除了可以在命令行中设置所有配置变量的值之外，用户也可以直接通过编辑配置文件的相关片段进行修改和设置，只需在文本编辑器中打开配置文件或者执行 `git config --edit` 命令即可。

在 Linux 系统中，一个刚初始化过的本地版本库配置文件和下列内容类似：

```
[core]
```

```
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
```

## 客户端基本配置

用户可以将 Git 配置选项分为两类：客户端的和服务端的。主要用于配置用户本地工作目录的设置属于客户端配置。服务端配置在本小节将会简要介绍，第 11 章将会详细介绍它们。这里向用户介绍的内容都是最基本的。

有很多支持配置的选项，不过需要用户设置的只有很少的一部分。其中大部分选项的默认值都是很合理的，显式设置它们只在某些极端情况下才会有用。有大量可供设置的选



项, 用户可以使用 `git config --help` 命令查看这些选项列表。这里涉及的只是最常见和最有用的若干选项。

`user.email` 和 `user.name` 这两个变量是切实需要用户配置的。

这些配置变量定义了作者的身份信息。当然, 如果用户正在给附注标签或者提交对象签名, 那么也许还需要配置用于签名的密钥 ID (详情可以参考第 5 章)。用户可以通过设置配置变量 `user.signingKey` 达到上述目的。

默认情况下, Git 会调用用户设定的任意系统级文本编辑器作为默认的编辑器 (使用环境变量 `VISUAL` 或者 `EDITOR` 进行设置; 不过前者只适用于图形化桌面环境), 用来创建或者编辑提交和标签的信息。Git 还会调用用户任意设定的寻呼机 (PAGER) 对 Git 命令的执行结果进行分页和浏览。为了修改默认设置, 用户可以对属性 `core.editor` 进行设置。相应的设置也包括 `core.pager` 属性。在上述属性为设置的情况下, Git 最终会退而求其次, 调用 `vi` 编辑器和更简单的寻呼机。



Git 中的呼叫器是自动调用的。默认的简易呼叫器不仅支持分页, 还支持增量查询。

此外, 使用默认配置的简易调用方式和 Git 采用 `LESS=FRX` 配置项的行为是类似的。这意味着当输出结果不足一页时, 系统就会自动跳过分页操作, 使用 ANSI 颜色代码, 并且不会清屏退出。

创建提交注释信息时还会受到 `commit.template` 的影响。如果用户设置了该配置变量, Git 将会在用户创建提交时使用该文件作为默认的注释信息。该模板一般不会和版本库一起安装。注意, 除非将 `commit.status` 的值设为 `false`, 否则 Git 将会把状态信息添加到提交注释模板中。

如果用户有一个注释信息策略, 那么这类模板会非常有用, 因为这可以大大增加采用该策略的概率。例如它可以使用已被注释掉的信息来填充注释信息。用户可以使用相应的钩子来检查提交相关注释信息是否符合该策略, 以便对上述方案进行功能扩展 (详情可以参考本章“提交过程钩子”部分的内容)。

工作区的文件状态会受到忽略模式和文件属性的影响 (详情可以参考第 4 章)。用户可以在自己项目树内的 `.gitignore` 文件中添加忽略模式 (通常 `.gitignore` 文件中标记了那些需要被跟踪的文件, 而其自身又被 Git 系统跟踪), 也可以在 `.git/info/excludes`

文件中将用户或者私人不感兴趣的文件排除。对于特定项目，有时用户还可以编写一个全局（单个用户）的 `.gitignore` 文件。在当前的 Git 软件中，可以使用 `core.excludesFile` 自定义上述文件的路径，而且系统给该文件提供了一个默认路径，即 `~/.config/git/ignore`。还有这类全局 `.gitattributes` 文件对应的配置变量 `core.attributesFile`，其默认路径是 `~/.config/git/attributes`。



实际上的路径是 `$XDG_CONFIG_HOME/git/ignore`，  
如果环境变量 `$XDG_CONFIG_HOME` 未设置或者为空，  
那么系统将会采用 `$HOME/.config/ git/ignore`。

虽然 Git 有一个内部的 `diff` 实现，用户可以通过变量 `diff.external` 配置外部工具取而代之。用户也许常常希望创建一个脚本包装器来帮助用户给 Git 传递参数，同时也可以在外部的 `diff` 工具需要传递相关参数时能够提供便利。默认情况下，Git 给 `diff` 程序传递参数的格式如下：

```
path old-file old-hex old-mode new-file new-hex new-mode
```

用户还可以参考图形化的 `diff` 和合并工具章节了解 `git difftool` 和 `git mergetool` 的相应配置。

## 为 pull 操作预设相应的变基和合并操作

默认情况下，执行 `git pull` 命令时，Git 将会通过合并操作将本地历史记录和拉取自远程版本库的历史记录进行串联。如果本地分支的历史记录是衍生自远程版本库的，那么还会创建一个合并提交。某些主张建议最好避免所有这类合并提交，通过变基来串联历史记录，以便达到最大限度地创建线性历史记录的目的（例如使用 `git pull--rebase` 命令），详情可以参考第 7 章。

有一些配置可以用来配置执行 `git pull` 命令时默认执行变基操作，以方便跟踪历史记录。此外还有配置选项 `pull.rebase` 和特定分支的 `branch.<name>.rebase` 选项可以达到上述目的，当然将它们设置为 `true` 时，其含义是告知 Git 在拉取更新过程中使用编辑替代合并操作（<name>分支是适用于后者）。它们还可以通过 `--preserve-merges` 选项来忽略变基操作，以便达到在变基过程中本地合并提交不被扁平化的目的。

用户可以通过 `branch.autoSetupRebase` 配置变量让 Git 在创建特定种类的新分支时，自动让每个分支执行“拉取数据然后变基”的操作。上述变量可供用户设置的值有 `never`、`local`（仅限本地跟踪分支）、`remote`（仅限远程跟踪分支）和 `always`（本地和远程跟踪分支）。

## 保留撤销信息-逾期对象

默认情况下，Git 将会自动删除未引用的对象，清理引用日志中的旧条目，打包松散的对象，以便达到减小版本库尺寸的目的。用户还可以执行 `git gc` 命令进行手动的垃圾收集。第8章已经向读者介绍过版本库的面向对象结构。

出于安全方面的考虑，Git 会把超过两周的未引用对象删除。不过用户可以通过 `gc.pruneExpire` 配置变量修改它：该设置通常是一个相对时间（例如 `1.month.ago`，用户可以使用点符号作为单词的分隔符）。为了禁用该过期时间（通常是通过命令行完成的），将会使用的参数值是 `now`。

默认情况下，可达修订对象分支外部引用的历史记录将会被保留 90 天（或者可以通过 `gc.reflogExpire` 进行设置）。不在当前历史记录中的引用日志实体，默认情况下将会被保留 30 天（也可以通过 `gc.reflogExpireUnreachable` 进行设置）。

上述两种设置都可以在单个引用名称的基础上进行配置，通过提供一个引用名称来匹配子段落名的模式，即 `gc.<pattern>.reflogExpire`。其他配置也可以进行类似的设置。

这还可以用来修改 `HEAD`、`refs/stash`（详情可以参考第4章），以及远程跟踪分支的 `refs/remotes/*` 的过期设置。该设置的值是一个时间段（例如 `6.months`），为了完全禁用引用日志中的过期时间，需要使用的参数值是 `never`。用户可以使用后者禁用暂存实体的过期功能。

## 格式化和空格

在协作开发过程中，代码格式化和空格导致的问题和潜在错误常常会令人沮丧，特别是在跨平台开发过程中表现得尤其严重。补丁和合并通常会潜在地引入空格变更，因为在 MS Windows、Linux 和 MacOS X 等不同操作系统环境下，编辑器会悄悄地引入这类变更和不同的换行符标记。Git 有一些配置选项可以帮助用户解决这类问题。

跨平台开发过程中一个非常重要的问题就是换行符的表示。这是因为文本文件在 MS Windows 中是采用一个回车符号（CR）和一个换行符号（LF）一起表示另起一行的，而同时 MacOS 和 Linux 系统只采用了一个换行符表示另起一行。很多在 MS Windows 中的文本



编辑器都会悄悄使用 CRLF 替换现有的 LF 格式换行符，或者使用 CRLF 表示另起一行，因此这就会引发一些潜在的，但是令人沮丧的问题。

Git 可以在用户将文件添加到索引时自动将换行符转换为 LF 来解决这个问题。如果用户的编辑器使用的换行符是 CRLF，那么当用户将代码签出到文件系统中时，也能自动将它转换成原生格式。有两个配置变量可以影响这个问题：`core.eol` 和 `core.autocrlf`。

第一个配置变量是 `core.eol`，它可以用来设置将文件签出到工作目录中时，文件类型是文本时，系统如何选用换行符（可以参考下一小节中“使用 `gitattributes` 配置单个文件”，并且可以简要回顾第 4 章介绍文件属性的相关内容）。

第二个配置变量是 `core.autocrlf`，它的资历更老一些，可以用来启用将换行符自动转换成 CRLF 的功能。将它的值设置为 `true` 之后，Git 会在用户签出文件时，将版本库中的换行符 LF 转换成 CRLF，反之亦然。这个设置对于使用 Windows 系统的用户来说可能是非常有用的。如果用户使用的是 Linux 或者 Mac 系统，那么该设置的用法也是类似的。为了禁用该功能，将版本库中换行符保持原貌，只需将该值设置为 `false` 即可。

这还可以用来处理部分空格问题，如换行符差异。导致该问题的主要因素是文件编辑。Git 也提供了不少用于检测和修复空格文件的办法。它可以查找和处理一些常见的空格问题。配置变量 `core.whitespace` 可以用来激活（默认禁用的功能）或者禁用相关功能（默认启用的功能）。有几种功能默认是启用的，它们分别是如下 3 种。

- `blank-at-eol`：用于查找行尾空格。
- `blank-at-eof`：用于查找文件尾部空行。
- `space-before-tab`：这会立即在文本行中 Tab 制表符缩进之前查找空格。

配置变量 `core.whitespace` 中的值 `trailing-space` 是 `blank-at-eol` 和 `blank-at-eof` 的简写形式。

有几个默认禁用的属性，但是可以将其启用的是如下 3 种。

- `indent-with-non-tab`：该选项会将文本行中使用空格代替 Tab 制表符的行为视为一个错误（根据制表符宽度判定）；该选项会强制使用 Tab 制表符表示缩进。
- `tab-in-indent`：监控文本行中使用制表符表示缩进的情况（这里使用制表符宽度修复这类空格错误）；本选项会强制使用空格表示缩进。
- `cr-at-eol`：该选项告知 Git 在行尾可以使用回车符作为换行符（允许用户在版本库中使用 CRLF 作为换行符）。



用户可以通过给配置变量 `core.whitespace` 设置以逗号分隔的参数值列表, 以便告知 Git 希望启用或者禁用哪些功能。为了禁用某个选项, 在该选项前添加 “-” 符号即可。例如用户只希望设置 `cr-at-eol` 和 `tab-in-indent` 两个选项, 以及制表符宽度为 4, 那么可以使用如下配置:

```
$ git config --local core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab,tabwidth=4
```

用户还可以使用 `whitespace` 属性对单个文件进行配置。例如用户可以使用该属性在测试用例中禁用空格检查来处理空格问题, 或者可以在 Python 2 源码文件中使用空格来表示缩进:

```
*.py whitespace=tab-in-indent
```

在用户执行 `git diff` 命令时, Git 将会检测这些问题, 并使用 `color.diff.whitespace` 设定的颜色将相关问题标记出来, 以便用户可以在创建新的提交之前将这些问题修复。在使用 `git apply` 命令应用相关补丁时, 用户既可以使用 `git apply --whitespace=warn` 命令, 要求 Git 向用户警示和空格有关的文件, 以及使用 `--whitespace=error` 选项, 将空格问题标记为错误; 也可以使用 `--whitespace=fix` 选项, 要求 Git 尝试自动修复这些问题。相应的配置也适用于 `git rebase` 命令。

## 服务端配置

Git 提供若干选项供用户进行服务端配置。与之有关的详情将会放在第 11 章具体展开。本小节将会对其中部分有趣的参数做一个简要介绍。

用户可以让 Git 服务端检查对象的一致性, 即在推送过程中将每个对象和接收到的 SHA-1 标识符进行匹配, 并通过一个布尔值配置变量 `receive.fsckObjects` 将它指向一个有效的对象。该操作默认是不启用的, 因为 `git fsck` 命令占用的资源比较大, 可能会拖慢整个操作, 特别是对于一个大型推送来说 (同时在一个大型版本库中)。这是一种预防故障和防范客户端恶意访问的检查机制。

如果某些被用户重写的提交对象被推送到了服务端 (这是一种糟糕的开发习惯, 详情可以参考第 8 章) 并尝试再次推送, 那么用户的操作将会被系统拒绝。客户端可能会在执行 `git push` 命令时使用 `--force` 选项强制推送更新到远程版本库。服务端可以通过将配置变量 `receive.denyNonFastForward` 设置为 `true`, 让 Git 拒绝接受客户端的强制推送操作。

`receive.denyDeletes` 变量用于处理 `denyNonFastForward` 策略的解决方案之

一是删除并重新创建一个分支。这会禁止系统删除分支和标签，因此用户如果想删除相关内容的话，必须在服务端手动删除相关引用。

这些特性还可以通过服务端类 `receive` 钩子实现，详情将会在本章“安装 Git 钩子”中进行介绍，同时还会在第 11 章对部分特性进一步扩展。

### 10.3.3 使用 `gitattribute` 配置单个文件

用户还可以专门针对某个路径声明一些自定义属性（也许是通过通配符），以便 Git 可以只对部分文件或者子目录应用这些设置。这些专门针对路径的设置被称为 `gitattribute`。

应用这类设置的优先级顺序是从 `$GIT_DIR/info/attributes` 文件中单个版本库本地路径设置开始的。接着是 `.gitattributes` 文件，从目标路径同名目录下的文件开始，然后回溯到父级目录的 `.gitattributes` 文件，依次类推，直到工作区顶层目录（项目的根目录）。最后是全局的单个用户属性文件（通过 `core.attributesFile` 变量指定，如果未指定，则会去 `~/.config/git/attributes` 目录查找）以及系统级的属性文件（默认安装时的目录是 `/etc/gitattributes`）。

第 4 章对可供用户选择的 Git 属性有详细介绍。此外，通过属性，用户还可以完成其他任务，例如，通过合并驱动为特定类型的文件声明特定的合并策略（例如变更日志），告知 Git 如何比较非文本文件，或者让 Git 在文件签入（在暂存变更和向版本库提交变更时，即在版本库数据库中创建对象）和签出（在写入文件系统时）过程中进行内容过滤。

#### Git 属性文件语法

一个 `gitattribute` 文件就是一个基于单个路径本地配置的简单文本文件。空行或者以 `#` 开头的行将会被忽略。

以 `#` 开头的行将会被当作程序代码注释，同时空行将被当作为提高可读性的分隔符。为了给某个路径声明一组属性，可以通过一组属性列表设定一个模式，并使用水平空格分隔它们：

```
pattern attribute1 attribute2
```

当路径匹配多个模式时，后面的模式行会覆盖前面模式行中的设置，其选取策略和 `.gitignore` 文件类似（用户可以将 Git 属性文件看作是系统级本地版本库文件，其优先级规则从简单到具体）。



Git 使用反斜杠作为模式的转义字符。例如模式是以#开头的,那么用户需要在第一个#前面添加一个反斜杠(即\#)。因为属性信息是由空格分隔的,模式中行尾的空格将被忽略,模式内部空格将被当作模式终结符号,除非它们被反斜杠引用转义(即“\”)。

如果模式不包含代表目录分隔符(/),Git 将会把该模式当作一个 shell 通配符模式,并搜索匹配相对于.gitattributes 文件路径(或者其他顶层属性文件)的路径名。例如模式\*.c 会匹配.gitattributes 文件中指定范围内的所有 C 程序文件。在模式开头添加斜杠将会匹配路径名起始位置。例如模式/\*.c 会匹配 bisect.c,但是不会匹配 builtin/bisect--helper.c,同时模式\*.c 会匹配上述两种情况。

如果模式中至少包含一个斜杠,Git 将会通过 fnmatch(3) 函数调用 FNM\_PATHNAME 标记,将之作为 shell 通配符处理。这意味着模式中模糊匹配不会包含目录分隔符,即路径名中的斜杠(/),匹配将会锚定路径开始的部分。例如模式 include/\*.h 会匹配 include/version.h,但是不会匹配 include/linux/asm.h 或者 libxdiff/includes/xdiff.h。Shell 通配符模糊匹配包括:\*匹配任意字符串(包括空字符串),? 匹配任意单个字符,[...]匹配任意词组(方括号内部\*和? 将会失去其代表的特殊含义)。注意,和正则表达式不同,字符类的取反/否定是通过!实现的,而非^。例如为了匹配任意不包含数字的字符串,用户可以使用 shell 模式[!0-9],它和正则表达式[^0-9]是等价的。

模式中两个连续的星号(\*\*)可能包含特殊的含义,但是这只适用于它们位于两个斜杠之间(\*\*/),以及位于斜杠和模式首尾之间的情况。在上述情况下,其含义是模糊匹配 0 个或者多个路径组件。因此,连续







的星号\*\*后面紧挨着一个斜杠，其含义是匹配所有目录，同时/\*\*表示匹配特定目录下任意文件或者子目录。每个属性对于给定路径可供选择的值有 4 种。第一，它可以是 set（属性是由表示 true 的特殊值设定的），这可以通过简单列出属性列表中的属性名来完成，例如 text。第二，它可以是 unset（属性是由表示 false 的特殊值设定的），这可以通过简单列出属性列表中的包含减号前缀的属性名来完成，例如 -text。第三，它可以设置为特殊值，可以简单地通过属性列表名跟随一个等号以及相关属性值进行设置，例如 -text=auto（注意，和配置文件语法不同的是，属性设定的等号两边不能包含任何空格）。第四，如果没有模式能够匹配该路径，或者该路径没有任何属性，那么就可以说该属性是未声明的（用户可以使用 !text 强制显式声明该属性是未声明的，来达到覆盖属性设置的目的）。

如果用户发现自己重复使用频率比较高的模式，可以考虑使用属性宏定义。虽然宏定义可以在本地、全局、系统级的属性文件中定义，但是它只能在顶层的 .gitattributes 文件中才能定义。宏是通过 [attr]<macro> 这样的格式来替换文件模式的，属性列表定义了宏的表达式。例如内置的二进制属性宏定义如下所示：

```
[attr]binary -diff -merge -text
```

## 10.4 Git 自动化钩子

通常有一些由特定先决条件产生的代码，它们既可以由自身触发执行，也可以通过外部工具强制执行。这类代码至少能够编译，并通过一个快速测试子集的测试。对于某些开发工作流，每个提交的信息也许需要引用一个问题 ID（或者 fix 修复信息模板），或者包含一个剥离签名行形式的原生数字签名。大部分情况下，这部分开发过程可以由 Git 自动完成。



和很多开发工具类似，当某个特定的重要预定义动作被执行时，即当某个特定事件被触发，Git 提供了一种能够在用户代码（自定义脚本）中执行用户自定义功能的机制。这类功能是被当作一个事件句柄触发的，其名为钩子。它允许用户添加一些额外的动作，至少对于部分钩子是如此，同时还可以停止或者触发其他功能的执行。

Git 中的钩子可以分为客户端钩子和服务端钩子两类。客户端钩子是通过本地操作触发的（在客户端），例如提交、应用一系列补丁、变基以及合并等。换句话说，服务端钩子是在服务器上执行的，即接收推送提交这类网络操作执行时被触发的。

用户还可以将这些钩子划分为 pre 钩子和 post 钩子。钩子被称为 pre 钩子的原因是它触发的时机通常是在某个操作执行完毕之前。如果相关操作的退出代码返回了一个非零值，那么它们将会取消 Git 当前正在执行的操作。post 钩子一般是在某个操作执行完毕之后触发的，可以用它来给用户发送通知或者添加日志记录，它们一般无法取消相关操作的执行。

### 10.4.1 安装 Git 钩子

Git 中的钩子是由若干可执行程序组成的（通常是脚本），它们一般存放在 Git 版本库管理区的 hooks/子目录下，对于非裸版本库来说，它们存放于 .git/hooks/目录下。每个钩子程序的命名都是和被触发的事件有关的，这意味如果用户希望在某个事件之后执行多个脚本，那么就需要自己对它们进行复用。

当用户通过 git init 命令初始化一个新的版本库时（该操作也可以通过 git clone 命令创建一个其他版本库的副本来完成，克隆操作会在内部调用 init 命令），Git 会使用一系列不活动的示例脚本填充钩子目录。其中很多脚本都是非常有用的，不过钩子的 API 文档中也对它们有详细的介绍。所有示例都是通过 shell 或者 Perl 脚本编写的，不过将它们命名为适当的可执行文件也能很好地完成任务。如果用户希望将这些钩子示例脚本打包使用，那么需要先对它们进行重命名，剥离文件名中的.sample 扩展，并确保它们拥有相应的可执行权限。

### 10.4.2 版本库模板

有时用户也许希望为所有版本库添加一组同样的钩子。为此用户可以创建一个全局（单个用户和系统级）的配置文件、一个全局属性文件和一个全局忽略列表。

事实证明，在创建版本库过程中选择需要填充的钩子是可行的。默认的示例钩子脚本是从 /usr/share/git-core/templates 中拷贝，然后填充到版本库的 .git/hooks 目录的。

此外，版本库创建模板的替代目标可以通命令行参数 --template 进行指定（对于 git clone 和 git init 命令来说），也可以通过环境变量 GIT\_TEMPLATE\_DIR 或者配置选项

`init.templateDir` (该选项可以在单个用户配置文件中设置) 指定。该目录必须遵循 `.git($GIT_DIR 目录下)` 的文件结构, 这意味着钩子需要被存放在上述目录中的 `hooks/` 子目录下。

注意, 这种机制也有一些不足之处。因为模板目录中的文件只是在 Git 版本库初始化和更新时进行拷贝的, 不会对现有的版本库造成影响。不过用户可以在现有版本中再次执行 `git init` 命令重新对它们进行初始化, 只需记得将任何对钩子的修改提前保存即可。



为一个开发团队维护钩子的工作是需要有点技巧的。一种可选方案是将用户自己的钩子存放在实际的项目目录中 (项目工作区中), 或者存放在一个独立的钩子版本库中, 如果有必要的话, 可以在 `.git/hooks` 目录下创建一个符号链接。甚至还有专门用于管理钩子的工具和框架, 用户可以在如下网址找到这类工具的应用案例: <http://githooks.com/>。

### 10.4.3 客户端钩子

有很多客户端钩子可供用户选择。它们大致可以划分为提交工作流钩子 (一组由创建新提交不同状态触发的钩子), 应用电子邮件工作流钩子, 以及其他钩子 (未整合到多钩子工作流中的钩子)。



需要着重强调的一点是, 用户在克隆版本库时并没有拷贝钩子。这么做的部分原因是出于安全方面的考虑, 因为钩子是自动运行, 并且大部分是不可见的, 用户需要手动拷贝 (重命名) 这些文件, 以便能够在创建或者重新初始化一个版本库时知道哪些钩子被安装了。这意味着用户不能依赖客户端钩子强制执行某个策略, 如果用户有这方面的硬性需求, 那么需要在服务端进行相应配置。

#### 提交过程钩子

在提交变更时, 有 4 个客户端钩子会被触发 (默认情况下), 它们分别是:

(1) 首先执行的是 `pre-commit` 钩子，即使之前用户调用了编辑器来输入提交的注释说明信息。它主要用来检查提交的快照，确认用户是否忘记了什么。该钩子返回一个非零值退出代码后，系统将会取消该提交操作。用户可以使用 `git commit --no-verify` 命令跳过触发该钩子的操作。该钩子不需要任何参数。

此外，该钩子还可以用于检查代码样式正确与否，执行静态代码分析，以便检查有问题的代码构造，确保代码能够顺利编译，并通过所有测试（新增代码需要执行覆盖测试），以及新增函数是否添加了相应的说明文档。该钩子默认会使用 `git diff --check` 命令（或者其他底层命令）检查空格错误（默认情况下是换行符空格），以及文件变更中非 ASCII 码的文件名变更。例如用户可以创建一个钩子，要求 Git 在用户提交一个包含脏工作目录状态的变更时向用户确认（工作区中的变更并非被提交变更的一部分），不过这是一种比较高级的技术。用户也可以尝试在新的钩子上添加是否在新的函数上添加了说明文档和单元测试的检查。

(2) `prepare-commit-msg` 钩子的执行是在默认提交说明信息创建之后（其中包括 `commit.template` 指定的给定文件的静态文本）、说明信息被编辑器打开之前。它允许用户编辑默认的提交注释信息，或者在提交者看到它之前以编程方式创建一个模板。如果该钩子以一个非零状态码退出，那么上述提交操作将会被取消。该钩子可以将文件路径作为参数来保留提交信息（后者是传递给编辑器的），以及和提交说明信息源有关的信息（该信息不会在普通的 `git commit` 命令中出现）：如果指定了 `-m` 或者 `-F` 选项，那么相关的内容是注释信息；如果给定的参数选项是 `-t` 或者设置了 `commit.template` 变量，那么是指模板；如果是合并提交或者存在 `.git/MERGE_MSG` 文件，那么操作对象是提交；如果存在 `.git/SQUASH_MSG` 文件，那么注释信息是和压缩有关的；如果信息是来自其他提交：指定了 `-c`、`-C` 或者 `--amend` 选项，那么操作对象是提交。在最后一种情况中，钩子还会获得额外的参数，即提交对象注释信息源的 SHA-1 码。

该钩子的主要用途是编辑或者创建提交注释信息，如果指定了 `--no-verify` 选项，那么该钩子将不会被触发。该钩子最有用的地方是用来处理自动生成的默认提交注释信息，例如模板化的提交信息、合并提交、压缩提交和已修改提交。Git 提供的示例钩子脚本中提供了对合并冲突的解释：它们被当作合并注释信息的一部分。

这类钩子的另外一个用途是通过 `branch.<branch-name>` 来描述当前给定分支。如果分支存在描述性信息，那么它将会被当作一个分支独立的动态注释模板。又或者检查当前用户是否在主题分支，并在项目问题跟踪记录系统中列出所有指派给用户的问题，以便用户可以更方便地添加合适的人造 ID 到注释信息中。



(3) `commit-msg` 钩子是在开发人员编写注释信息之后、提交对象实际写入版本之前执行的。它需要指定一个参数，即用户提供的包含注释信息的临时文件路径（默认是 `.git/COMMIT_EDITMSG`）。

如果执行该脚本后返回了一个非零值状态码，那么 Git 会取消相关的提交操作，因此用户可以用它进行相关验证，例如提交信息匹配项目的状态，或者提交信息是否符合相应的请求模式。Git 提供的示例钩子脚本可以用来检查、排序和移除重复的已剥离签名信息——相关签名行（如果剥离签名会导致连锁反应，那么该签名可能不是用户希望使用的）。用户可以放心地使用该钩子检查引用的问题代码是否正确（也许还要对它们进行扩展，为每个提及的问题添加当前的摘要信息）。

Gerrit 代码审核软件（可能还需要安装到本地版本库）提供一种提交信息钩子来自动创建、插入和维护唯一的变更 ID：在 `git commit` 命令执行过程中附着到剥离签名的行上。该行可以用来跟踪一个提交的迭代记录，如果修订中的提交注释信息推送到 Gerrit 时缺乏这类信息，服务端将会向用户提供如何获取和安装该钩子脚本的操作指南。

(4) `post-commit` 钩子在实体提交过程执行完毕后触发。它不需指定任何参数，不过此时可以将提交操作过程中得到的修订作为 `HEAD`。该钩子的退出状态码将会被忽略。

一般来说，该脚本（和其他大部分 `post-*` 脚本一样）经常用于通知用户和记录日志，并且它并不会影响 `git commit` 命令的输出结果。用户可以使用它来触发一个在类似 Jenkins 的持续集成工具中的本地编译操作。大部分情况下，用户希望使用 `post-receive` 钩子在专用的持续集成服务器上完成该任务。

该钩子的另外一个用途是列出所有任务计划、代码中修复记录注释信息和说明文档有关的信息（例如作者、版本号、文件路径、行数以及注释信息等），将它们打印传递到标准输出钩子，以使用户制作备忘录，并能够及时更新它们。

## 电子邮件补丁钩子

用户可以为基于电子邮件的工作流创建 3 种钩子（提交是通过电子邮件发送的）。它们都是通过 `git am` 命令触发的（该命令的命名来自 `apply mailbox`），它可以用来处理包含补丁的电子邮件（例如使用 `git format-patch` 命令创建补丁，以及使用 `git sent-email` 命令发送补丁），并将它们转换成一系列的提交。这些钩子有以下几种：

(1) 首先执行的是 `applypatch-msg` 钩子。它会在从补丁中提取注释信息之后、应用补丁自身之前执行。一般来说，对于一个非 `post-*` 钩子，如果该钩子返回了一个非零值状态码，那么 Git 将会取消应用补丁。它需要提供一个执行参数，其包含被提取注释信



息的临时文件名。

用户可以使用该钩子来确认提交信息是否被正确格式化了，或者对被脚本修改过的提交信息文件进行格式化。Git 提供的示例 `applypatch-msg` 钩子脚本会简单地调用 `commit-msg` 钩子——当然是在该钩子存在的情况下（该脚本文件存在并且是可以执行的）。

(2) 接下来要执行的钩子是 `pre-applypatch`。它在补丁被应用到工作区之后、提交对象被创建之前触发。用户可以用它来查看创建提交对象之前的项目状态，例如运行测试。如果该钩子返回一个非零值状态码，那么系统将在不提交补丁的情况下取消 `git am` 命令的执行。

Git 提供的示例钩子将会简单地执行 `pre-commit` 钩子。

(3) 最后将要执行的钩子是 `post-applypatch`，它在提交对象创建之后被触发。它可以用来通知用户和记录日志，例如通知所有开发人员或者补丁的作者，用户已经应用了该补丁。

### 其他客户端钩子

还有其他一些类型的客户端钩子，它们并不适合处理单个过程中的一系列步骤。

`pre-rebase` 钩子是在用户执行任何变基操作之前触发的。和所有 `pre-*` 钩子类似，它可以通过返回非零值状态码取消变基操作。用户可以使用该钩子撤销对已发布提交的变基（重写）操作。该钩子是通过基础分支（被提取的一系列上游节点）和被变基的分支进行调用的。只有在被变基的分支非当前分支时才需要传递第二个参数。Git 提供的 `pre-rebase` 钩子示例代码将会尝试执行上述操作，不过该脚本会假定特定的 Git 项目开发工作可能和用户的工作流不匹配（注意，编辑提交的同时也会重写该提交，变基操作可能会创建一个分支的副本来替代重写它们）。

`pre-push` 钩子是在执行 `git push` 命令过程中执行的，即签出远程状态之后（确认哪些修订记录是服务端没有的）、推送任意变更之前。该钩子是通过远程版本库引用（URL 或者远程版本库名称）和实际的推送 URL（远程版本库地址）作为脚本参数进行调用的。被推送提交有关的信息将会以标准输入的形式提供，逐行对每个引用进行更新。用户可以使用该钩子在执行推送前对一组引用更新进行验证，返回非零值状态码将会取消该推送操作。安装的示例脚本只会简单地检查准备推送的修订记录中是否存在正在研发的提交对象，或者提交注释信息中包含 `nopush` 关键字的提交。如果存在上述情况，该钩子就会取消该推送操作。用户甚至可以通过一个钩子向用户弹出对话框确认是否希望执行相关操作。该钩子是为了辅助服务端校验检查，以便避免在数据传输过程中因为无法通过校验而导致中断。

`post-rewrite` 钩子是在相关命令重写历史记录之后执行的（替换提交对象），例如 `git commit --amend` 和 `git rebase` 命令。不过需要注意的是，该钩子不会在诸如 `git filter-branch` 这样的大型历史记录重写操作之后执行。触发重写的命令类型是通过单个参数进行传递的，同时重写列表将会以标准输入的形式发送。该钩子的大部分用途和 `post-checkout`、`post-merge` 等钩子类似，它可以在自动拷贝笔记之后被触发执行，并且可以通过配置变量 `notes.rewriteRef` 进行控制（和笔记有关的详情可以参考第 8 章）。

`post-checkout` 钩子是在 `git checkout` 命令（或者 `git checkout<file>` 命令）更新工作树之后触发的。该钩子需要指定 3 个参数：上一修订和当前 `HEAD`（可能和实际稍有不同）的 `SHA-1` 码，以及声明是否签出整个项目（如果用户正在编辑分支，那么信号参数的值是 1）或者单个文件（从索引或者居民提交获取文件，信号标记的值是 0）的信号标记。有一种特殊情况，那就是在 `git clone` 命令执行完毕之后、初始化签出过程中，该钩子将会把全为零的 `SHA-1` 码作为第一个参数进行传递（作为源修订版本）。用户可以使用该钩子建立符合自己需求的工作目录。这也许还意味着需要在版本库之外处理大量的二进制文件（它可以作为单个 Git 属性过滤器的替代方案），因为这些文件可能是用户不希望存放在版本库中的。用户也可以为工作目录的元数据设置相应的属性，例如完全读写权限、所有者、分组、时间、扩展属性以及 `ACL` 等。它还可以用来执行版本库的有效性检查，或者通过自动显示与上一签出版本之间的差异（或者只是差异统计）对 `git checkout` 命令进行功能扩展（假定它们之间存在差异）。

`post-merge` 钩子是在一个合并操作成功执行之后触发的。用户可以像使用 `post-checkout` 钩子那样，利用该钩子存储 Git 不会跟踪的工作树中的元数据和其他数据，例如包含完整读写权限的数据（或者直接调用 `post-checkout` 钩子即可）。该钩子可以用来校验 Git 系统外部的数据，方便用户在工作树发生变更时将外部数据拷贝到 Git 系统中。

对于 Git 来说，版本库中的对象都是不可变的（例如表示修订的提交对象）；重写历史记录（甚至修改提交对象）的过程实际上是创建了一个被修改过的对象拷贝，然后切换到该对象上，同时丢弃了之前的历史记录。删除一个分支之后，同时相应的历史记录也会被丢弃。为了防止版本库体积不至于急速膨胀，Git 偶尔会通过移除旧的未引用对象进行垃圾收集工作。在以前的大部分历史版本中，这一操作是通过 Git 系统执行 `git gc --auto` 命令实现的。`pre-auto-gc` 钩子就是启动垃圾收集工作之前被触发的，并且它可以用来终止该操作。例如用户遇到电池电力不足的情况时。它还可以用来通知用户垃圾清理工作开始了。

## 10.4.4 服务端钩子

除了运行在用户本地版本库上的客户端钩子之外，还有两个非常重要的服务端钩子，它们是系统管理员用来给用户项目执行任意强制策略的。这些钩子是在用户将变更记录推送到服务端之前和之后触发的。**pre** 钩子返回一个非零值状态码时将会终止一个推送操作或者其中的一部分操作。通过 **pre** 钩子打印的说明信息将会被发送到客户端。用户可以使用这类钩子配置复杂的推送策略。**gitolite** 和 **Git** 托管服务这样的 **Git** 版本库管理工具，会采用这些策略实现对版本库更精细的访问控制。**post** 钩子可以用来给用户发送通知、启动编译过程（或者只是重新编译和部署），或者执行一套完整的用例测试，例如可以将它们作为持续集成解决方案的一部分。

在编写服务端钩子脚本时，用户需要考虑版本库操作序列中的每个过程都发生了什么，哪些相应的信息是可用的，既可以作为参数传递，也可以作为标准输入。

下面是服务端接收一个推送时的大致操作流程：

（1）简单来说，第一步是所有存在于客户端但是不在服务端的对象将会被发送到服务端并存储起来（不过还没有添加索引）。如果接收上述对象的操作没能正确执行（例如由于硬盘空间不足），那么整个推送操作将无法完成。

（2）接下来执行的是 **pre-receive** 钩子。它会获取标准输入中被推送引用的描述列表。如果它返回一个非零值的状态码，那么系统将会取消整个推送操作，并且不会接受任意一个被推送的引用。

（3）对于每个将被更新的引用，相关步骤如下：

① 系统内置的完整性校验失败后会拒绝将变更推送到引用，其中包括检查签出分支的更新或者非快进式推送（除非是强制性的）等。

② **update** 钩子将会通过被推送的引用作为参数触发。如果该脚本返回一个非零值状态码，那么该引用会被拒绝，因为示例钩子将会阻止未添加注释的标签进入版本库。

③ 该引用会被更新（除非在现代的 **Git** 系统中，要求推送操作必须是原子提交）。

（4）如果该推送是原子提交，那么所有引用都将会被更新（假定没有被拒绝的情况发生）。

（5）**post-receive** 钩子将会通过获取和 **pre-receive** 钩子获取的相同数据继续执行。该钩子可以用来更新其他服务（例如通知持续集成服务器）或者通知其他用户（通过电子邮件、邮件列表、IRC 或者 bug 跟踪系统）。



(6) 每个被更新的引用都会触发 `post-update` 钩子。它还可以用来记录日志。示例钩子会执行 `git update-server-info` 命令来预置一个版本库，保存用于哑传输协议的额外数据。不过如果先执行一遍 `post-receive` 钩子，它可以运行得更好。

(7) 如果推送尝试更新当前签出的分支，并且配置变量 `receive.denyCurrentBranch` 的值是 `updateInstead`，那么 `push-to-checkout` 钩子将会被触发执行。

用户在使用 `pre` 钩子时需要注意，此时引用还没有更新，`post` 钩子无法影响操作结果。用户可以使用 `pre` 钩子处理访问控制（权限检查），`post` 钩子可以用来处理用户通知、数据更新和日志记录等。

用户将会在第 11 章了解采用 Git 强制策略的示例钩子，还会学习其他工具使用这类钩子的方法，例如访问控制和推送过程中触发相应的动作。

## 10.5 Git 扩展

Git 为用户提供了一些可以对它进行功能扩展的机制。用户可以添加快捷方式，创建新的命令，为新的传输协议添加支持。上述所有操作都不需要修改 Git 的源代码。

### 10.5.1 Git 命令行别名

有一个小技巧可以帮助用户更方便、快捷地使用 Git 命令行，即 Git 别名。理论上来说创建一个别名非常容易，用户只需创建一个配置变量 `alias.<命令名称>` 即可，其值是别名的表达式。

别名的用途之一是为用户常用的 Git 命令和参数创建缩写。另外一种用法是创建新的 Git 命令。下面是几个与此有关的示例：

```
$ git config --global alias.co checkout
$ git config --global alias.ci commit
$ git config --global alias.lg log --graph --oneline --decorate
$ git config --global alias.aliases 'config --get-regexp ^alias\.'
```

上述命令中前面的内容是和输入有关的，例如 `git ci` 和输入 `git commit` 命令等价。别名可以像正常的 Git 命令那样接收参数。Git 并没有提供任何默认的别名用于为普通的操作命令定义快捷方式，除非用户使用的是由 Felipe Contreras 开发的更人性化版本的 Git 程序 `git-fc`。

参数是通过空格进行分隔的，同时还支持常见的 shell 引用和转义符号。具体来说，用户



可以在单个参数中使用一对双引号 ("a b") 或者反斜杠 (a\ b) 对其中的空格进行转义。



不过需要注意的是, 用户不能添加和 Git 命令同名的别名。换句话说, 用户不能使用别名来改变 Git 命令的行为。该规定背后的原因是, 这样会导致现有的脚本和钩子出现异常行为。隐藏在 Git 命令 (和 Git 命令同名的别名) 背后的别名将会被自动忽略。

有时用户可能需要运行一个外部命令, 而不是一个使用别名表示的 Git 名。又或者用户希望将若干独立命令的执行结果串联。在这种情况下, 用户可以在别名前面加上 ! 符号 (带感叹号):

```
$ git config --global alias.unmerged \
'!git ls-files --unmerged | cut -f2 | sort -u'
```

因为这里别名的表达式的第一个命令可以是一个外部工具, 因此用户需要像前面示例中那样显式声明 git 包装器。



#### 注意:

在很多 shell 程序中, 例如 bash, ! 是历史记录扩展表达式字符, 因此需要使用反斜杠对它们进行转义, 例如 \!, 或者使用单引号将它包裹起来。

注意, 这类 shell 命令可以在版本库的顶层目录执行 (在切换到顶层目录之后), 而不一定必须在当前目录执行。Git 可以通过环境变量 GIT\_PREFIX 设置当前目录相对于版本库顶层目录的路径, 即通过执行 git rev-parse --show-prefix 命令。一般来说, 这里可能会用到 git rev-parse 命令 (以及其他 git 包装器选项)。

前面提及的事实可以用于创建别名。别名 git serve 会执行 git daemon 命令, 以只读方式获取 git://127.0.0.1/服务上的当前版本库信息, 并且可以充分利用别名中的 shell 命令是自版本库顶层目录执行的这一事实:

```
[alias]
serve = !git daemon --reuseaddr --verbose --base-path=. --export-all
./.git
```

有时, 用户需要对参数进行重排, 使用一个参数两次, 或者将一个之前在管道中的参

数传给相关命令。用户也许还希望像\$1、\$2 这样依次类推的方式引用后续的参数。用户可以在先前的示例中找到一个技巧，即执行 shell 命令时搭配-c 选项参数，就像第一个示例之后提及的那样。最后的破折号表示参数是以\$1 开始的，而非\$0。一种更常见的语法是定义并马上执行一个 shell 函数，就像第二个示例中表示的那样（这是一个更好的方案，因为它少了一个引用层级，允许用户使用标准的 shell 参数进行处理）：

```
[alias]
record-1 = !sh -c 'git add -p -- $@ && git commit' -
record-2 = !f() { git add -p -- $@ && git commit }; f
```

别名也集成了命令行自动补全功能。在确定自动补全哪个别名时，Git 将会搜索对应的 git 命令，跳过开放的括号或者单引号（这同时支持前面提到的两种语法）。在当前流行的 Git 中（版本为 2.1 或者更高），用户可以使用空指令“:”来声明预期的自动补全样式。例如别名表达式 !f() { : git commit ; ... } f 将会为 git commit 命令执行自动补全功能，不管别名其他部分如何。

Git 别名还集成了助手系统，如果用户在别名中使用了 --help 选项，Git 将会告知用户它的表达式（以使用户能够查看相关的帮助文档）：

```
$ git co --help
'git co' is aliased to 'checkout'
```

## 10.5.2 添加新的 Git 命令

别名最适合处理小型的单句，并将它们转换成一组有用的 Git 命令。用户可以编写复杂的别名，不过在处理大型脚本时，用户也许更希望直接将它们集成到 Git 中。

Git 的子命令可以独立执行，并在 Git 执行路径上继续存续（用户可以执行 git --exec-path 命令查找它）；在 Linux 系统下，该路径一般是 usr/libexec/git-core。git 可执行体自身就是一个瘦包装器，它能够感知子命令的存续位置。如果 git foo 命令不是一个内置的 Git 命令，包装器会首先在 Git 执行路径上进行搜索，然后在环境变量 \$PATH 中搜索。后者使得在不访问系统空间的情况下编写本地 Git 扩展（本地 Git 命令）成为可能。

这个特性也让 Git 项目中其他部分或多或少集成了用户接口，例如 git imerge（详情可以参考第 7 章）和 git annex（详情可以参考第 9 章），它也是 Git Extras 这类项目提供额外的 Git 命令的基础。

不过需要注意的是，如果用户没有为自己的命令在常用位置安装相关的说明文档或者配置文档系统，以便能够查找到与之相关的帮助说明，那么 git foo --help 命令将无

法正常工作。

### 10.5.3 凭据助手和远程版本库助手

还有另外一个地方可以简单地安装可执行体，就能对 Git 进行功能扩展。远程版本库助手被 Git 调用是在它需要和远程版本库交互，以及遇到 Git 原生不能兼容的远程传输协议的时候。用户如果希望了解详情，可以参考第 5 章。

当 Git 遇到 `<transport>://<address>` 格式的 URL 地址，其中 `<transport>` 是一个 Git 原生不支持的（伪）协议时，它会自动调用 `git remote-<transport>`，并将远程版本库名称或者 URL 作为该命令的参数，不过有时也会将 `<address>` 作为第二个参数来替代 URL 地址。此外，使用 `remote.<remote-name>.vcs` 设置 `<transport>` 时，Git 将会显式执行 `git remote-<transport>` 命令访问远程版本库。

Git 中的助手机制是和使用格式规范的外部脚本交互有关的。

每个远程版本库助手都被预期可以支持一组子命令。用户可以在 `gitremote-helpers(1)` 帮助页面找到创建新的助手程序这个问题的详细信息。

Git 中还有另外一种助手，其名为凭据助手。它可以帮助 Git 获取用户必需的凭据，例如通过 HTTP 协议访问远程版本库。虽然它们是通过配置文件声明的，就像 `merge` 和 `diff` 驱动程序一样，以及清洁和涂抹过滤器。

## 10.6 小结

本章为用户介绍了所有高效使用 Git 的必备工具。读者已经了解了如何更方便地使用命令行接口，以及使用 Git 动态命令提示符、命令行自动补全、Git 命令自动更正和配色等功能，提高 Git 的使用效率；还学习了现有的替代性接口应用，其范围从替代性底层命令到图形化客户端变体。

读者需要在使用多种方式改变 Git 命令的行为时多加留意；同时还了解了 Git 如何访问自身的配置，以及配置变量子集的查询；继而学习了如何使用钩子实现 Git 相关任务的自动处理和如何充分使用它们；最后学习了如何使用新的命令和兼容的 URL 架构对 Git 进行功能扩展。

本章的主要任务是帮助用户更高效地使用 Git，第 11 章将会介绍如何让 Git 能够更高效地为其他开发人员服务。用户将会进一步了解服务端钩子的详情以及它们的具体使用，同时还将学习如何维护版本库。



# 第 11 章

## Git 日常管理

上一章已经介绍过如何使用 Git 钩子自动化完成某些工作任务。其中详细介绍了客户端钩子的使用，同时对服务端钩子只做了简要介绍。本章将会介绍服务端钩子的具体使用，以及客户端钩子的辅助作用。

前面章节的内容主要是帮助用户作为开发者、团队协作成员以及维护者来使用 Git 的，本书谈及创建版本库和分支结构时，是从 Git 用户的角度出发的。

本章将会站在 Git 管理员的角度来帮助读者。其中包括配置远程 Git 版本库以及它们的访问控制。其中的内容涵盖了让版本库更好地运作的工作（即版本库维护），以及定位和修复版本库中的错误。本章还会介绍一些传输协议，以及如何使用服务端钩子实现某些强制开发策略。此外，读者还会了解到若干可以用来管理版本库的工具以及它们的优缺点，方便用户选择。

本章的主要内容包括以下几个方面。

- 服务端钩子——实现一个策略和通知。
- 传输协议、身份验证和授权。
- Git 的服务端配置。
- 管理远程版本库的第三方工具。
- 使用轮替和命名空间降低托管版本库的大小。
- 通过签名推送断言引用更新和启用审核功能。
- 改进服务端性能和辅助初始化克隆。
- 版本库故障检测和修复。



- 使用引用日志和 `git fsck` 修复错误。
- Git 版本库维护和打包。
- Git 开发工作流扩展。

## 11.1 版本库维护

有时，用户也许需要对版本库做一些清理工作，通常这么做的原因是希望版本库结构可以更紧凑一些。在将一个版本库从其他版本控制系统迁移过来之后，这类清理工作将是非常必要的。

现代的 Git 系统会在每个版本库中循环往复地执行 `git gc --auto` 命令。该命令会检查系统中是否存在过多松散对象（对象是作为独立文件存储的，每个对象一个文件，而非存放在一个压缩文件中；对象几乎都是以松散结构创建的），如果存在上述情况，系统会启动垃圾收集操作。垃圾收集意味着将松散对象聚拢到一起，并将它们放到压缩文件中，或者将一些小型压缩文件添加到一个大的压缩文件中。

此外，系统还会将引用打包到 `packed-refs` 文件中。不可达对象，甚至包括引用日志和旧的对象，将不会被重新打包。之后 Git 将会删除这些松散对象和被重新压缩的压缩文件（是根据某些松散对象文件寿命的安全冗余进行压缩的），这样就达到了清理旧的不可达对象的目的。在 `gc.*` 命名空间下，还有不少可以控制垃圾收集操作的配置选项。

用户可以手动输入 `git gc --auto` 命令执行自动垃圾收集操作，也可以使用 `git gc` 命令强制执行该操作。`git count-objects` 命令（可能还需要用到 `-v` 选项）可以用来检查是否出现了需要进行重新打包操作的迹象。用户甚至可以通过 `git repack`、`git pack-refs`、`git prune` 和 `git prune-packed` 命令单独执行垃圾收集单个操作步骤。

默认情况下，Git 将会尝试复用以前的打包操作结果，以便减少重新打包对 CPU 时间的占用，同时还能很好地提高硬盘空间的利用率。在某些情况下，用户优化版本库尺寸所需花费的时间代价更小：通过 `git gc --aggressive` 命令，这一想法是可行的（或者通过手动执行 `git repack`，并辅以相应的参数，对版本库进行重新压缩）。建议用户在从其他版本控制系统导入 Git 版本库时执行上述操作；Git 会采用该机制优化导入数据的执行速度（快速导入流），而非最终的版本库尺寸。

还有一些因为 `git gc` 命令无法处理的版本库维护问题，这是由它的原生特性导致的。其中之一就是清理（删除）远程版本库中已经被删除的远程跟踪分支。这个问题可以通过

`git fetch --prune`、`git remote prune` 以及在单个分支上执行 `git branch --delete --remotes <远程跟踪分支>` 命令来解决。

该操作是留给用户去处理的，而非 `git gc` 命令，因为 Git 无法轻易地获知哪些基于用户工作成果的远程跟踪分支需要被清理。

## 11.2 数据恢复和故障诊断

工作中永远不犯错几乎是不可能的，这对于使用 Git 来说也是如此。本书介绍的知识 and 用户过去曾经使用 Git 积累的经验，可能会帮助用户减少犯错的概率。不过需要注意的是，Git 会尽最大的努力尝试保留用户的工作成果不致丢失，很多错误都是可以恢复的。

### 11.2.1 恢复已丢弃的提交记录

用户偶尔丢失某个提交的情况是可能发生的。例如用户因为失误强制删除了某个不正确的分支，现在用户又需要用到它们；又或者用户将某个分支回退到了一个错误的位置；也可能用户在执行相关操作时处于一个错误的分支上。

假定上述某种情况发生了，那么是否有亡羊补牢的机会，拿回用户自己的提交对象呢？

因为 Git 并不会马上删除对象，而是会将它们保留一段时间，只有在它们经过垃圾收集检查之后、变成不可达对象时才会删除它们，用户丢失的提交对象有可能就在其中，所以用户需要做的就是找到它们即可。如前所述，垃圾收集操作拥有自己的安全机制，因此如果用户找到了想要的对象，只需将它们恢复，同时最好使用 `git config gc.auto never` 命令临时禁用自动垃圾收集功能。

通常情况下，找到并恢复丢失的提交记录的方法是使用 `git reflog` 工具。对于每个分支和被分离的 HEAD，Git 会自动记录（日志）用户本地版本库分支外部引用的位置和时间，以及该对象被创建的原因。这种记录就叫引用日志（reflog）。

每次用户创建提交或者回退分支时，分支和 HEAD 的引用日志都会随之更新。每次用户修改分支后，HEAD 引用日志也会一起更新。

用户可以通过 `git reflog` 或者 `git reflog <branch>` 命令随时查看分支外部引用的状态。此外还可以使用 `git log -g` 命令达到上述目的，`-g` 选项是 `--walk-reflog` 的简写。这可以为用户输出一个普通的可配置日志结果。另外，`--grep-reflog=<pattern>` 选项可以用来搜索引用日志：

```
$ git reflog
6c89dee HEAD@{0}: commit: Ping asynchronously
d996b71 HEAD@{1}: rebase -i (finish): returning to refs/heads/ajax
d996b71 HEAD@{2}: rebase -i (continue): Ping asynchronously WIP
89579c9 HEAD@{3}: rebase -i (pick): Use Ajax mode
7c6d322 HEAD@{4}: commit (amend): Simplify index()
ele6f65 HEAD@{5}: cherry-pick: fast-forward
eea7a7c HEAD@{6}: checkout: moving from ssh-check to ajax
c3e77bf HEAD@{7}: reset: moving to ajax@{1}
```

读者对第2章介绍的<ref>@{<n>}这类语法应该还有印象。根据引用日志中的信息，用户可以将有问题的分支回退到一组操作之前的版本，或者从列表中任意一个提交对象开始创建一个新的分支。

现在假定用户因为失误删除了一个分支而导致数据丢失。引用日志因为自身实现的原因（例如名为 foo 的分支的引用是 refs/heads/foo，它在引用日志中对应的文件是 .git/logs/refs/heads/foo），给定分支在被删除时，其对应的引用日志也会被一并删除。

也许用户仍然拥有 HEAD 在引用日志中的必要信息，除非用户在不涉及工作区的情况下操作了分支外部引用，但是想要找到它们并非易事。

在引用日志中不存在相关信息的情况下，查找必要信息以便找回恢复丢失的提交对象的唯一办法是使用 git fsck 命令，它会完整地对用户版本库进行检查。通过 --full 选项，用户可以使用该命令找到所有未引用的对象：

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (58/58), done.
dangling commit 50b836cb93af955ca99f2ccd4a1cc4014dc01a58
dangling blob 59fc7435baf79180a3835dddc52752f6044bab99
dangling blob fd64375c1f2b17b735f3145446d267822ae3ddd5
[...]
```

从上述查询结果中可以看到，包含 SHA-1 标识符的未引用（丢失）提交对象位于 commit 字符串前缀的信息行中。为了找到所有未引用的提交对象，用户可以使用 grep "commit" 对 git fsck 命令的输出结果进行过滤，使用 cut -d' ' -f3 对它们的 SHA-1 码进行提取，然后将这些修订作为参数传给 git log --stdin --no-walk 命令，进行二次查询。

```
$ git fsck --full | grep "commit" | cut -d' ' -f3 | git log --stdin --no-
```



walk

## 11.2.2 Git 故障诊断

`git fsck` 命令的主要用途是检测版本库中是否存在损坏的数据。除了可以用来查找过期对象之外，它还会对每个对象进行完整性校验，并追踪它们的可达性。它可以发现损坏和丢失的对象；如果损坏的数据仅限于用户的版本库克隆，而正确的版本可以在其他版本库中找到（例如备份或者其他归档文件），用户可以尝试从一个未损坏的数据源中恢复这些对象。

不过有时错误的原因可能更复杂。也许用户可以在团队之外找到一个 Git 专家，不过版本库中的数据却是项目特有的。最低限度地重现问题并不一定总能奏效。对于现代 Git 来说，如果问题是结构化的，用户可以在重现问题的同时，尝试使用 `git fast-export --anonymize` 命令将相关数据从版本库中剥离。

如果版本库没什么问题，问题出在 Git 的操作上，那么用户可以尝试将各种跟踪和除错机制集成到 Git 系统内部，或者也可以尝试增加命令的冗余度。

用户可以通过相应环境变量启用跟踪机制。通过设置相应环境变量的值为 1、2 和 true，跟踪结果可以被写入标准的异常流中。其他介于 2 到 10 之间的整数值将被用于表示打开记录跟踪结果的文件描述符。用户还可以将这类环境变量的值设置成存放跟踪结果信息文件的绝对路径。

和跟踪有关的变量包括以下几个（查阅 git 包装器的说明文档可以看到完整的列表）：

- `GIT_TRACE`：这会启用一般的跟踪信息，并且不是针对任意特定类目的。其中包括 Git 别名表达式（详情可以参考第 10 章）、内置 Git 命令的执行和外部命令的执行（例如 `pager`、`editor` 和 `helper`）。
- `GIT_TRACE_PACKET`：这会对“智能”传输协议的网络操作启用数据包级的信息跟踪。这还可以帮助用户跟踪网络传输协议的问题或者用户创建的远程版本库服务器的故障。对于浅克隆的版本库，其调试和拉取对应的变量是 `GIT_TRACE_SHALLOW`。
- `GIT_TRACE_SETUP`：这会启用跟踪信息，打印版本库管理区、工作区和当前工作目录位置信息和前缀（版本库目录结构下最后的一个子目录）。
- `GIT_TRACE_PERFORMANCE`：这会显示所有 Git 命令的执行总时间统计信息。

此外还有 `GIT_CURL_VERBOSE`，它用于发送通过 `curl` 库记录 HTTP 网络操作后生成的信息；`GIT_MERGE_VERBOSE` 用于控制递归合并策略生成的操作结果的统计信息。



## 11.3 Git 服务端配置

前面的章节为用户掌握处理大部分日常的版本控制工作提供了足够的知识。第 5 章向用户介绍了如何为团队协作选择版本库的架构。本章将会向读者介绍如何配置 Git 远程版本库服务。

Git 版本库管理的主题涵盖范围广泛。有不少专门介绍特定的版本库解决方案的书籍，类似的方案包括 Gitolite、Gerrit、GitHub 和 GitLab。接下来的内容将会对用户选择这类解决方案或者构建自己的初步架构大有裨益。

首先会从管理远程版本库自身的工具和机制开始，然后介绍几种发布 Git 版本库服务的方法（将版本库添加到服务器上）。

### 11.3.1 服务端钩子

被服务端调用的钩子可以用来进行服务端管理。此外，这些钩子还可以通过执行授权验证控制远程版本库的访问，并且可以确保添加的版本库中的提交符合最低的安全规范。后者还需要借助客户端钩子才能达到最佳效果，客户端钩子的详情可以参考第 10 章。这样用户就不会只在希望发布提交时，才会收到他们的提交未通过审核的通知了。换句话说，客户端钩子实现的验证机制可以很容易通过 `--no-verify` 选项跳过（因此服务端钩子验证就很有必要了），而且用户需要记得安装相关的组件。



不过需要注意的是，服务端钩子只会在推送过程中被调用，用户需要在拉取（克隆）操作过程中提供其他的方案实现访问控制。

很明显，在使用哑传输协议时也不会调用钩子，因为服务端没有安装 Git 程序可以调用它们。

在编写钩子实现某些 Git 强制性策略时，用户需要注意被调用钩子处于哪个阶段、哪些信息是可用的。知道传递给钩子的相对信息也是非常重要的，不过用户可以在 Git 帮助文档的 `githook` 章节方便地找到与之相关的最新信息。前一章已经对服务端钩子做了一个简要介绍，本章将会进一步深入讨论这个主题。

所有的服务端钩子都是通过 `git receive-pack` 命令进行调用的，它主要是负责接收将要发布的提交的（顾名思义，它是以打包文件的形式接收这些提交的）。对于每个钩子，除了 `post-*` 这类之外，如果钩子是以一个非零值状态码返回的，那么相关操作将会被终止，并且不会有进一步的状态转换。`post` 钩子是在相关操作执行完毕后触发的，因此不存在操作中断的情况。

标准的输出和标准的异常输出结果都会被转发给客户端的 `git send-pack`，因此该钩子可以方便地通过打印这类信息将它们传递给用户（如果该钩子是通过 `shell` 脚本编写的，那么可以使用 `echo` 语句输出）。

注意，在所有钩子完成它们的所有操作之前，客户端不会断开连接，因此如果用户尝试执行某些比较耗时的操作时需要特别留意，例如自动化测试。最好在启动这类耗时操作异步和退出时提供一个钩子，使得客户端能够完成相关操作。

用户需要注意的是，在执行 `pre` 钩子中，系统还没有更新引用，该 `post` 钩子不能影响操作结果。用户可以使用 `pre` 钩子进行访问控制（权限检查），使用 `post` 钩子进行消息通知，更新端点数据和日志记录等。钩子会在操作序列中被一一列出。

## pre-receive 钩子

首先执行的钩子是 `pre-receive` 钩子。它是用户在远程版本库上更新引用（分支、标签和笔记等）之前，接收完毕所有对象之后触发的。数据接收操作一经完成，它就被触发了。如果服务端接收拟发布对象失败了，例如因为硬盘空间不足或者没有提供正确的权限，那么整个 `git push` 操作将会在 Git 触发上述钩子之前被中断。

该钩子不需要指定任何参数，所有的信息通过标准的输入脚本传递即可。对于每个将被更新的引用，它接收的数据行格式如下：

```
<old-SHA1-value> <new-SHA1-value> <full-ref-name>
```

新建的引用将会包含值为 40 个零的旧 SHA-1 码，同时将被删除的引用会包含和新建引用等价的新 SHA-1 码。同样的转换也适用于其他地方，即钩子接收旧的和被更新引用的新状态。

如果更新未被接受，那么该钩子可以快速地帮助用户检测这一状态，例如当接收到的提交并没有遵循特定的策略或者签名推送无效时（稍后会介绍详情）。注意，在使用它进行访问控制时，用户（例如验证授权）需要获得某些授权令牌，将之与 `getpwuid` 命令或者 `USER` 这样的环境变量一起使用。这还有赖于服务端设置和配置环境。

## 推送非裸版本库的 push-to-update 钩子

在向非裸版本库推送提交时，如果该推送尝试更新当前签出的分支，那么将会执行 push-to-checkout 操作。如果配置变量 `receive.denyCurrentBranch` 的值为 `updateInstead`（替代下列某个参数值：`true`、`refuse`、`warn`、`false` 或 `ignore`），那么将自动执行上述操作。该钩子接收到的 SHA-1 码是提交的标识符，即当前将被更新分支的外部引用。

引入该机制的目的是处理同步工作目录时某一端不能很好地进行交互式访问（例如通过 SSH 交互式访问），或者作为一种简单的部署架构。它可以用来部署一个在线网站，或者在不同操作系统之间执行代码测试。

如果该钩子不存在，当 Git 发现工作区或者索引的状态和 HEAD 不同，即状态是“不干净”的，那么就会拒绝执行该更新操作。该钩子还可以用于重写默认行为。

用户可以巧妙地使用该钩子修改工作区和索引，将它们迁移到理想的目标状态。例如，为了切换到新分支的外部引用并同时保留本地变更（`-u` 选项会更新工作区的文件），执行 `git read-tree -u -m HEAD "$1"` 命令即可（`-m` 选项会对两个提交/树对象执行快进式合并）。如果该钩子退出时返回了一个非零值，那么它会拒绝推送当前签出的分支。

## update 钩子

接下来将要执行的是 `update` 钩子，它会在每个引用被更新时分别被调用。该钩子的触发是在非快进式签出之后，并且单个引用内置的完整性检查可以通过 `receive.denyDeletes`、`receive.denyDeleteCurrent`、`receive.denyCurrentBranch` 和 `receive.denyNonFastForwards` 进行配置。

注意，该钩子以非零值退出时将会拒绝更新相关引用，如果该推送是原子的，那么任意引用被拒绝更新后都会中断整个推送操作。对于普通的推送操作，只有单个被更新的引用会被拒绝执行更新操作，推送过程中的其他引用将会如常处理。

该钩子会接收和被更新引用有关的信息作为参数，其顺序为：被更新引用的全名，推送前存放于引用中的旧 SHA-1 对象名，推送后存放于引用中的新 SHA-1 对象名。

示例钩子 `update.sample` 可以用于屏蔽源版本库未注释的标签，同时还允许或禁止删除和修改标签，删除和创建分支。该示例钩子的所有可配置选项都可以通过 `hooks.*` 配置变量完成，这比硬编码的方式好很多。

`contrib/hooks/` 中还有很多功能强大的、处理更新的 Perl 脚本，它们可以用作演示该钩子进行访问控制的示例。该钩子还可以配备一个外部的配置文件，例如用户可以配置



相应访问权限，使得系统只接受特定作者生成的提交和标签，以及其他作者能够获得正确的访问权限。

很多版本库管理工具，例如 Gitolite，都采用了该钩子来完成相应任务。如果用户由于某些原因希望早于这类工具提供的钩子执行之前调用自己的钩子，那么需要事先阅读该工具的帮助手册。

## post-receive 钩子

所有引用被更新之后，将会执行 `post-receive` 钩子。它接收的数据和 `pre-receive` 钩子是一样的。只有到了这个阶段，所有引用才会指向新的 SHA-1 码。这有可能是在某个引用被更新后，其他用户又对它进行了修改，但是该钩子能够在它之前进行验证。该钩子还可以用于更新其他服务（例如通知持续集成服务器），通知用户（通过电子邮件、邮件列表、IRC 聊天频道或者一个工单跟踪系统），以及用于推送审计的日志信息。它可以用来替代 `post-update` 钩子。

目前没有默认的 `post-receive` 钩子，不过用户可以在 `contrib/hooks` 中找到一个简单的 `post-receive-email` 脚本以及它的替代品 `git-multimail`。

这两个示例钩子实际上是各自独立于 Git 本身开发的，不过是为了方便起见，将它们和 Git 源码放在一起一并提供了。`git-multimail` 会为每个变更的引用发送一个电子邮件摘要，为每个新增提交附加变更细节（作为回复）发送相应的引用变更电子邮件，以及为每个新增的附注标签发生通知邮件。它们都可以分别配置相应的电子邮件地址和附件，同时还可以分别将这些信息包含到电子邮件中。

以第三方工具 `irker` 为例，它提供的脚本可以用于 Git 的 `post-receive` 钩子通过 `irker` 守护进程（需独立配置）将新增变更的通知发送到相应的 IRC 频道。

## post-update 钩子（传统机制）

接下来执行的是 `post-update` 钩子。每个引用被成功更新实际上是通过将它的名称作为其中的一个参数完成的，该钩子可以接受若干参数。这也只是部分信息，用户无法知道被更新引用旧的（原有）的值和新的值（目标值）具体是什么，当前引用非常容易处于竞态条件下（如前所述）。因此，如果用户确实需要获知引用的位置，那么 `post-receive` 钩子是一个更好的解决方案。

示例钩子将会通过执行 `git update-server-info` 命令创建和保留额外的信息，为哑传输协议预备一个版本库。如果准备发布该版本库，或者希望通过 HTTP 或者其他基

于人力的传输的方式对它进行拷贝和发布，那么用户最好考虑启用它。不过，在时下流行的 Git 软件中，将 `receive.updateServerInfo` 的值设置为 `true` 就完全可以满足需要，因此该钩子不再是必需的了。

## 11.3.2 使用钩子实现 Git 强制策略

实现强制性策略的具体方式是通过服务端钩子完成的，即 `pre-receive` 或者 `update` 钩子。如果用户希望处理单个引用，那么就会用到后者。客户端钩子可以用来帮助开发者留意该策略，不过也可以禁用、跳过和不启用它们。

### 服务端钩子的强制策略

某些开发策略可能会要求每个提交的注释信息格式必须遵循特定的模板。例如，某些用户也许希望每个非合并提交注释信息中包含原生数字签名的签发格式——行，或者通过包含类似引用的字符串 2387 这样的格式来将每个提交和 bug 跟踪系统关联起来。可能性可以说是无限的。

为了实现这样一个钩子，用户首先需要将引用的新旧值转换成所有被推送提交的列表（用户既可以在 `pre-receive` 中逐行读取，也可以将它们作为 `update` 钩子的参数）。对于某些极个别情况需要特别留意：删除一个引用（无可推送的提交）、创建一个新引用、可能出现的非快进式推送（用户需要使用合并基底作为更基础的修订区间，例如使用 `git merge-base` 命令）、推送标签、推送笔记，以及其他非分支推送。将一个修订区间转换成一组提交列表可以通过 `git rev-list`（底层）命令完成，与之等价的、面向用户的命令是 `git log`（高层命令）。默认情况下，该命令将只会按行输出特定修订区间中提交的 SHA-1 码以及其他信息。

对于每个修订来说，用户需要获取相关的注释信息，并检查它们是否符合特定的模板策略。用户还可以使用另外一个底层命令，即 `git cat-file`，通过跳过第一个空白行之前的内容提取它的注释信息。该空白行是为了分隔原生注释体中的元数据注释信息的：

```
$ git cat-file commit a7b1a955
tree 171626fc3b628182703c3b3c5da6a8c65b187b52
parent 5d2584867fe4e94ab7d211a206bc0bc3804d37a9
author Alice Developer <alice@example.com> 1440011825 +0200
committer Alice Developer <alice@example.com> 1440011825 +0200
```

```
Added COPYRIGHT file
```

或者用户可以使用 `git show -s` 或者 `git log -1` 这两个高层命令替代使用 `git cat-file` 命令。不过，用户还需要声明精确的输出格式，例如 `git show -s --format=%B <SHA1>`。

用户获取这些注释信息之后，就可以使用正则表达式或者其他工具对每个注释信息进行匹配校验，以便确认它们是否符合相关的策略规范。

另外一部分策略可以是对分支管理的限制。例如用户希望阻止长期开发分支的删除（详情可以参考第 6 章），同时允许删除主题分支。为了识别它们，以便确保被删除的分支是否是一个主题分支，用户既可以添加一组可配置的分支列表进行管理约束，也可以假定所有主题分支都使用 `<user>/<topic>` 这样的命名约定。后一种方案还可以强制要求系统新建主题分支时遵循相关策略，以便匹配该命名约定。

可以想见的是，用户可以在主题分支没有被合并的情况下，专门创建一个快进式策略，不过为实现这个策略的校验过程将是比较特殊的。

一般来说，项目中只有某个特定用户拥有推送官方版本库变更的权限（即所谓的提交位）。通过服务端钩子，用户可以配置版本库允许任何人推送变更，不过仅限于特定的分支目录，所有其他目录的推送权限都将受到限制。

用户还可以使用服务端钩子让版本库只接收附注标签，给该标签签名的公钥位于特定的公钥服务器中（这样一来，可以方便其他开发人员检验），并且该标签不能被删除或更新。如果有必要的话，用户可以限制签名标签只能来自特定范围内（并且可以配置）的若干用户，例如强制策略可以声明只有一个维护者可以标记项目的预览版本号（通过创建相应的具名标签，例如 `v0.9`）。

### 客户端钩子的异常预警策略

拥有一个严格的强制性开发策略并不是一个好的解决方案，并且还不能提供一种方法来帮助用户监控和履行上述策略。在推送过程中拒绝某人的推送是令人沮丧的；为了修复某个问题阻止用户发布提交，用户就不得不编辑它的历史记录，详情可以参考第 8 章。

解决这个问题的办法是提供一些方便用户安装的客户端钩子，当用户违反了某个开发策略时，Git 能够马上自动通知用户，使他们能够及时发现自己生成的变更会被服务端拒绝接受。这样做的目的是尽可能在提交变更之前及早修复错误。这些客户端钩子必须以某种方式分发，因为钩子在版本库克隆时并没有一起被拷贝。分发这类钩子的具体方法可以参考第 10 章。

如果说变更的内容有什么不足的话，那么也许是某些文件只能由某些特定开发者进行



编辑，警告信息可以通过预提交对象完成。`prepare-commit-msg` 钩子（和配置变量 `commit.template`）可以帮助开发者在编写提交注释信息时使用自定义模板进行填充。用户还可以让 Git 对注释信息进行检查，即通过 `commit-msg` 钩子在提交被记录之前完成上述操作。该钩子将会找到并提示用户，提交对象和注释信息的格式是否正确，相关策略指定的必要信息是否已经都包含到提交中了。该钩子还可以用来替代或者辅助 `pre-commit` 钩子检查用户是否修改了管理员不允许修改的文件。

`pre-rebase` 钩子可以用来校验用户有没有尝试以某种方式重写历史记录，继而导致了非快进式推送（在服务端使用 `receive.denyNonFastForwards`，强制推送操作将无法执行）。

作为最后的手段，`pre-push` 钩子可以在系统尝试连接远程版本库之前对本地版本库进行完整性检查。

### 11.3.3 签名推送

第 5 章已经介绍过几种机制，通过它们，开发人员可以确保自己工作成果的完整性和真实性：签名标签、签名提交和签名合并（合并签名标签）。所有这些机制都要求这些对象（以及其中包含的变更）来自签名者。

不过签名标签和提交并没有要求开发者能够通过特定分支的外部引用访问特定修订版本。托管网站的身份验证不能方便地进行事后审计，这需要用户信任托管网站和它的权限验证机制。目前的 Git 程序（2.2 版本以上）允许用户为此对推送进行签名。

签名推送需要用户在服务端对配置变量 `receive.certNonceSeed` 进行设置，同时客户端采用 `git push --signed` 命令。处理签名推送是由服务端钩子完成的。

客户端发送的签名推送证书在版本库中是以 `blob` 对象的形式存在，并且是通过 GPG 算法进行校验的。

`pre-receive` 钩子可以查看若干 `GIT_PUSH_CERT_*` 环境变量（详情可以参考 `git-receive-pack` 钩子的帮助页面），从而决定接受或者拒绝给定签名推送。

用于审计的签名推送日志可以通过 `post-receive` 钩子进行记录。用户可以将签名推送信息作为一封电子邮件通知发送，也可以将推送信息附加到一个日志文件中。推送操作的签名证书中包含一个标识符，其中包括客户端 GPG 密钥、版本库 URL 地址，以及在分支或标签上执行相关信息，它能够以相同格式作为 `pre-receive` 和 `post-receive` 钩子的输入。

## 11.3.4 Git 版本库服务

第 5 章我们已经介绍过 4 种 Git 常用的连接远程版本库的协议：本地协议、HTTP 协议、SSH 协议（安全外壳）和 Git 原生协议。这些内容是以客户端访问远程版本库的视角进行阐述的，讨论了这些协议的具体作用，以及远程版本库支持多种协议时该如何选择。

本章将会以管理员的视角解释如何进行后续配置，对 Git 版本库进行调整，继而能够兼容不同传输协议。这里我们还会对每种协议进行检查，让读者了解它们是如何进行身份验证和授权的。

### 本地协议

这是最基本的协议，客户端使用版本库路径或者像 `file://` 这样的 URL 地址即可访问远程版本库。用户只需拥有一个共享文件系统，例如 NFS 或者 CIFS，其中包含 Git 远程版本库即可。如果用户已经拥有了访问网络文件系统的权限，那么这是一个不错的方案，因为用户不需要再搭建任何服务器了。

使用基于文件的传输协议访问版本库时，其访问权限是由现有文件权限和网络访问权限一同控制的。

用户为了拉取和克隆版本库数据，需要获得读取权限，推送操作需要获得写入权限。

对于后一种情况，如果用户希望启用推送，那么最好以这种方式建立一个版本库，以防推送操作对相关权限产生不良影响。这可以通过 `git init` 命令（或者 `git clone` 命令）创建一个版本库时附加 `--shared` 选项来实现。该选项允许属于同一组的用户使用精确的组 ID 执行推送操作，以便确保所有组成员都能操作版本库。

这种方法的缺点是：和普通的网络访问和配置相应的服务器相比，搭建一个支持共享访问的网络文件系统难度较大，并且兼容多个远程脚本库版本库地址对安全性也有一定要求。通过互联网安装远程版本库磁盘有一定难度，并且效率不佳。

该协议并不能保证版本库不会受到意外损害。每个用户对版本库内部文件都有完整的访问权限，缺少避免版本库受到意外损害的防护机制。

### SSH 协议

SSH（安全外壳）是比较常见的（特别是 Linux 用户）自托管版本库服务的传输协议。SSH 访问在大部分情况下已经被当作一种安全地登录远程主机的方式。假如服务器还不支持，那么它的安装和配置也非常简单。SSH 是一种支持权限验证和数据加密的网络传输协议。

换句话说，管理员不能通过 SSH 协议让客户端对相关版本库进行匿名访问。客户端通过 SSH 协议访问某个主机时权限是受限的。该协议不支持对公共版本库的匿名只读访问。

一般来说，有两种方法可以通过 SSH 协议访问 Git 版本库。第一种是为每个尝试访问版本库的用户在服务端创建一个独立账号（当然该账号的权限是有限的，并且不需要完整的 shell 访问权限。这种情况下，用户可以为特定 Git 账号使用 `git-shell` 作为他们的登录 shell）。这种方式既支持普通的 SSH 访问（即用户提供密码），也支持公钥登录访问。在一个账号对应一个用户的情况下，访问控制方面的情况和本地协议类似，即名义上访问是由文件系统权限控制的。

第二种方法是创建一个独立的 shell 账号，一般就是针对 Git 用户的，特别是访问 Git 版本库和使用公钥登录的授权用户。每个拥有访问版本库权限的用户，将会需要发送其公钥给管理员，管理员会把这些密钥添加到已授权密钥列表中。实际的用户是通过密钥进行识别验证，继而登录服务器的。

另外一种替代性方案是从 LDAP 服务器获得 SSH 服务端授权验证或者其他中心式验证架构（通常是为了实现单点登录）。一旦客户端获得了 shell 访问权限（有限的），就可以使用任意 SSH 认证机制了。

## Git 匿名协议

接下来是 Git 协议。它是通过一个特定的简单 TCP 守护程序提供服务的，并且会监听一个专门的端口（默认是 9148 端口）。这是一种为 Git 版本库提供快速匿名只读访问服务的常用解决方案。

Git 协议的服务端是 `git daemon`，其配置相对比较容易。一般来说，用户需要以守护进程的形式运行该命令。如何运行该守护进程（服务器）取决于用户使用的操作系统。它可以是一个 `upstart` 脚本、一个 `systemd` 单元文件或者一个 `sysvinit` 脚本。常用的解决方案是使用 `inetd` 或者 `xinetd`。

用户可以使用 `--base-path=<directory>` 选项重新映射版本库请求的给定相对路径（Git 版本库的项目根目录），同时它还为虚拟主机提供了支持，详情可以参考 `git-daemon` 的说明文档。默认情况下，`git daemon` 将会只输出在 `gitdir` 目录中的 `git-daemon-export-ok` 文件；如果想输出所有文件，可以通过 `--export-all` 选项实现。一般来说，用户还会希望启用 `--reuseaddr` 选项，它允许在连接超时的情况下无须等待，直接重新启动相关服务。

Git 协议的不足之处在于缺乏授权验证机制和运行在比较少见的端口上（这要求用户在



防火墙上网开一面), 缺乏授权验证机制是因为默认情况下它只是用于读取访问的, 即拉取和克隆版本库。

一般来说它需要搭配 SSH (永远需要授权, 不允许匿名访问) 或者 HTTPS 协议才能执行推送操作。

用户可以通过设置让它支持推送 (通过命令行选项 `--enable=<service>` 启用 `receive-pack` 服务, 或者在单个版本库上将配置变量 `daemon.receivePack` 的值设为 `true`), 不过一般不建议这么做。唯一可以帮助钩子实现访问控制的信息是客户端地址, 除非用户要求所有推送都必须经过签名。用户可以在一个访问钩子中执行外部命令, 不过这并不能提供更多和客户端有关的信息。



`upload-archive` 文件归档服务可能也是用户希望启用的, 与之相关的命令是 `git archive --remote`。

缺乏验证机制意味着不仅 Git 服务端不知道谁访问了版本库, 而且客户端必须假定通过网络访问服务器时不存在网络地址欺诈。该传输是未经加密的, 因此所有内容都是以明文形式传输的。

## 智能 HTTP (S) 协议

配置所谓的“智能”HTTP (S) 协议时会执行一段服务端脚本, 并在服务端调用 `git receive-pack` 和 `git upload-pack` 命令。Git 提供了一个名为 `git-http-backend` 的 CGI 脚本处理此任务。该 CGI 脚本可以检测客户端是否能够识别智能 HTTP 协议, 如果客户端不能识别, 它将会使用哑协议进行传输 (向下兼容性)。

为了使用此协议, 用户需要某种 CGI 服务器, 例如 Apache (使用此服务器之后, 用户还需要启用 `mod_cgi` 模块或者其等价物, 以及 `mod_env` 和 `mod_alias` 模块)。参数传递是通过环境变量实现的 (因此, 使用 `mod_env` 是为了应对使用 Apache 的情况): `GIT_PROJECT_ROOT` 指定了版本库的位置所在, 可选变量 `GIT_HTTP_EXPORT_ALL` 是为了处理当用户希望导出所有版本库文件, 而非仅限于 `git-daemon-export-ok` 文件时。

验证机制由 Web 服务器完成。特殊情况下, 用户可以设置允许不需授权的匿名只读访问服务, 同时对推送操作进行授权验证。和 SSH 协议类似, 使用 HTTPS 协议时可以进行数据加密和服务端授权验证。使用 HTTP (S) 协议时, 拉取和推送操作相关的 URL 地址是一样的。用户还可以进行一定的配置, 使得用户浏览 Git 版本库的 Web 接口和拉取操作使用一样的 URL 地址。



git-http-backend 帮助文档中包含了一些为了应对不同情况而搭建 Apache 环境的说明信息，其中还包括未授权读取和未授权写入。它稍微有一些复杂，因为初始化引用广播用到了查询字符串，同时 receive-pack 服务请求会用到路径信息。换句话说，读取和写入所需的有效账户的授权验证和不受限制的写入，对于服务端钩子来说更简单，通常也是更容易让人接受的解决方案。

如果用户尝试向一个需要授权验证的版本库执行推送操作，服务端会要求用户输入验证凭据。因为 HTTP 协议是无状态的，并且有时会包含多个连接，那么使用验证助手来避免在单次操作中重复输入密码或者不得不在本地硬盘上（或者是远程版本库 URL 地址中）存放密码信息就非常有用。

## 哑协议

如果用户不能在服务端运行 Git 软件，那么仍然可以使用哑协议，因为它并不需要 Git 软件。哑 HTTP (S) 协议会将 Git 版本库当作 Web 服务器上的普通静态文件处理。不过，为了能够使用这种协议，Git 需要另外将文件 objects/info/packs 和 info/refs 存放于服务端，并使用 git update-server-info 命令定期更新它们的信息。该命令经常用于通过前述智能协议执行推送操作（默认的 post-update 钩子会执行该操作，如果将 receive.updateServerInfo 的值设为 true，那么 git-receive-pack 钩子也会执行该操作）。

使用哑协议也能执行推送，不过这需要在服务端做一些额外配置，以便支持用户通过特定协议上传文件。对于哑 HTTP (S) 传输协议，这意味着需要配置 WebDAV。

权限验证是通过将之当作 Web 服务的静态文件来实现的。很明显，对于这种传输来说，Git 的服务端钩子并没有被触发，因此它们不能用作进一步的访问权限控制。



### 注意：

对于当前的 Git 软件来说，哑传输协议是采用远程版本库助手的 curl 族特性实现的。默认情况下，用户可能并没有安装它们。

该协议通过下载请求的引用（文本文件）来完成相关操作（例如拉取），从中查找引用的提交对象（因此，所需的服务端信息文件至少对其中的对象进行打包压缩），获取它们，

然后遍历整个修订记录链条，查找所需的每个对象；如果相关对象不存在于本地版本库，则自动下载它们。如果版本库所需的修订区间上的对象没有进行良好的打包整理，那么该遍历过程将会非常低效。它需要创建大量的连接并且下载整个压缩包，即使其中只包含一个所需的对象也是如此。

使用智能协议后，Git 的客户端和服务端之间可以进行协商，以便确定需要发送哪些对象（需要/已存在的对象信息交换）。然后 Git 通过两边对象的分布情况创建一个自定义压缩包，其中包含的只是双方的增量信息，即两边的差异对象信息。另外一边重写接收到的压缩包进行自包含操作。

### 远程版本库助手

Git 允许用户通过编写远程助手程序兼容新的传输协议。这种机制也可以用于兼容外源版本库。Git 和一个版本库交互时，需要远程版本库助手生成一个独立的子进程作为其助手程序，通过在该子进程上的一组命令作为标准输入和输出来达到和版本库交互的目的。

用户还可以了解到第三方远程版本库助手如何兼容版本库访问的新方法，例如 `git-remote-dropbox` 采用 **Dropbox** 服务来存储 Git 远程版本库。不过请注意，远程版本库助手的特性可能会受限于 Git 内置的传输协议。

## 11.3.5 Git 版本库管理工具

目前来看，用户不需要自己编写 Git 版本库管理工具。市面上有大量的第三方解决方案可供用户选择。这些工具数不胜数，甚至专家提供的建议也存在风险。Git 生态系统的发展非常活跃，当时最好的工具可能在撰写本文时已经被其他工具替代了。

这里重点是管理员介绍工具的种类，如果希望了解 GUI 工具的详情，可以参考第 10 章的内容。

首先，有不少 Git 版本库管理的解决方案（我们已经在 `contrib/area` 中的智能更新脚本中看到这样一个例子）。这些工具主要聚焦于访问控制，通常对于授权验证部分会竭力帮助用户方便地添加版本库，以及提高对其访问权限管理的便捷性。比较典型的工具有 **Gitolite**。它们提供的某些机制还允许用户添加属于自己的访问控制约束条件。

其次，还有不少 Web 接口，它们可以让用户通过 Web 浏览器查看 Git 版本库信息。使用 Web 接口甚至可以创建一个新的修订。它们之间的差异主要在功能上，不过通常最少会向用户提供一个可用的 Git 版本库列表、每个版本库的摘要视图（其功能与 `git log` 和 `git show` 命令等价），以及版本库内部文件列表。类似这样的工具有使用 Perl 编写的 `gitweb` 脚本，



并且该脚本是通过 Git 进行分发的。另外一款工具是 [https:// www.kernel.org/](https://www.kernel.org/) 为了管理 Linux 内核版本库（以及其他版本库）而开发的 `cg`。

比较常用的是代码审核工具（代码协作）。这使得开发者通过 Web 接口查看每个人提交的变更历史记录成为可能。这些工具还允许用户创建新的项目和进行访问控制管理。类似的工具有 Gerrit 代码审核工具。

最后，还有不少 Git 版本库托管解决方案，它们通常通过 Web 接口为用户提供版本库管理服务，并且允许用户添加用户、创建版本库、管理访问权限，以及其他可以通过浏览器完成的 Git 版本库日常管理工作。类似这样的工具有 GitLab。还有不少类似的源代码管理系统，它们将版本库托管服务、团队协作和开发管理进行了有机整合。类似的工具有 Phabricator 和 Kallithea。

当然，用户不需要自行托管自己的代码。有大量的第三方托管服务可供选择，如 GitHub、Bitbucket 等。甚至还有使用开源托管管理工具的托管服务，例如 GitLab。

### 11.3.6 版本库托管应用技巧

如果用户希望使用自托管的 Git 版本库服务，那么下面是一些提高服务端性能和用户满意度方面的建议。

#### 版本库瘦身

如果用户托管了多个同一版本库的派生（fork）副本，那么也许会希望通过某种共享对象的方式减少它们对硬盘空间的占用。一种办法是使用替换机制，同时创建一个派生（例如使用 `git clone --reference` 命令）。在这种情况下，派生版本库在自身未发现相关对象时，会自动查找其父对象的存储引用。

不过这种情况下存在两个问题。首先，用户需要确保借用自其他版本库的对象被当作替换对象存储时不会消失（在借用版本库中）。这可以通过在版本库中将借用版本库引用和借出对象进行链接来实现，例如在 `refs/borrowed/` 命名空间下。

其次，出现在借用版本库中的对象不可以进行重复记录自动清理操作。为此用户需要执行 `git repack -a -d -l` 命令，其内部会将 `--local` 选项传递给 `git pack-objects` 命令。

一个替代性的解决方案是在单个版本库中保留每个派生记录，然后在修订 DAG 上使用 `gitnamespaces` 分别对这些独立视图进行管理，一个视图对应一个派生。对于简易 Git 程序，该方案意味着版本库是通过普通对象存储的 URL 进行编址的，并且命名空间会使用特定的派生。一般来说，这是通过服务端配置或者版本库管理工具进行管理的。这种机制可

以将版本库地址转换成普通的版本库和命名空间。`git-http-backend` 帮助文档中包含了一个在单个版本库中从不同命名空间配置多个版本库服务的示例。`Gitolite` 还为命名空间提供了逻辑关系的支持，不过并不是所有特性都能够兼容逻辑版本库。

将多个版本库以命名空间的形式存放于单个版本库中可以避免存储同一对象的重复拷贝。它可以在不需要日常维护的情况下，自动阻止新对象之间产生重复数据，这一点和替换方案恰恰相反。换句话说，它的安全性更弱一些，用户需要假定任何人都可以访问独立的命名空间，也就是说在版本库中，任何人都可以访问其他命名空间，虽然这可能并不是管理员的本意。

### 使用位图压缩提高智能协议效率

另外一个可能会遇到的问题是用户使用自托管版本库时智能协议的性能问题。对于访问服务器的客户端，快速完成相关操作是非常重要的。因为作为管理员来说，并不希望因为提供了 Git 版本库服务而导致服务器 CPU 负载过大。

JGit 的特点之一就是能够明显改善统计对象数目过程的性能，同时使用这种技巧从版本库对外提供对象访问服务。该特性是一个位图索引文件，支持 Git 2.0 及以上版本。

该文件和压缩文件及其索引是存放在一起的，既可以通过执行 `git repack -A -d --write-bitmap-index` 命令手工生成，也可以通过设置配置变量 `repack.writeBitmaps` 的值为 `true`，继而和打包文件一起自动生成。该方案的缺点是位图会占用额外的硬盘空间，初始化重新打包时需要花费额外的时间创建位图索引。

### 大型非复用初始化克隆问题

包含大型代码基和悠久历史记录的版本库的体积可以变得非常庞大。问题在于初始化克隆操作，用户可能需要克隆大型版本库的所有内容，这是一个要么全盘接受、要么什么也得不到的操作。至少对于现代的（安全和高效）智能传输协议是如此：`SSH`、`git://` 和智能 `HTTP(S)`。如果网络连接环境不太可靠，这可能会成为一个问题。目前还不支持克隆复用，并且不幸的是，它对于 Git 软件开发者来说是一个根本性的难题。不过这并不意味着托管服务管理员在帮助用户进行初始化克隆操作方面就无能为力了。

一种办法是使用 `git bundle` 命令创建一个可以用于初始化克隆的静态文件，或者将之作为初始化克隆的版本库引用（如果用户的 Git 软件版本在 2.3 以上的话，可以通过 `git clone --reference=<bundle> --dissociate` 命令完成相关操作）。这个 `bundle` 文件可以使用任意协议进行传输，特别是对于 `HTTP`、`FTP`、`rsync` 和 `BitTorrent` 等协议来说，它们还支持断点续传。采用此约定的人员，除了在开发者文档中要解释如何获得这样一个

bundle 文件之外,既可以使用相同的 URL 访问版本库,也可以通过访问 .bundle 文件(代替空的扩展或者 .git 后缀)实现。

还有更高级的方法,例如逐步深入地浅克隆(或者只是使用 `git clone --depth 1` 命令获得必需的数据)以及使用 GitTorrent 这类软件。

## 11.4 改进开发 workflow

版本控制只是开发 workflow 中的一小部分,此外还有工作管理、代码审核和验证、执行自动化测试、生成编译等步骤。

其中大部分步骤可以借助特定工具完成,并且很多都提供了对 Git 集成的支持。例如代码审核可以使用 Gerrit 管理,这需要每个变更在公开发布之前都必须通过代码审核。另外一个示例是搭建开发环境,以便推送变更到公共版本库时,用户在问题跟踪系统中可以根据提交信息中的模式自动关闭工单(记录 bug 编号)。这还可以通过服务端钩子或者托管服务的 Web 钩子来完成。

一个版本库还可以当作一个网关使用,以便用户执行自动化测试(例如可以借助 Jenkins/Hudson 持续集成服务),并且确保只有通过所有测试才发布这些变更。其他版本库可以配置成为多种支持系统执行编译过程。很多工具和服务都支持推送并发布这一机制(例如 Heroku 和 Google 的 App 引擎)。

Git 可以自动通知用户和开发者有新增变更发布。其形式可以通过电子邮件、邮件列表或 IRC 频道,以及基于 Web 的公告板应用。可供选择的方式很多,需要用户做的就是找到它们。

## 11.5 小结

本章的内容涵盖了在使用 Git 过程中和日常管理有关的多个问题。读者学习了 Git 的日常维护、数据恢复和版本库故障诊断技巧,还学习了如何在服务端配置 Git、如何使用服务端钩子、如何管理远程版本库。本章介绍的技巧可以帮助远程版本库获得更好的性能。本章的内容将会帮助用户选择合适的 Git 版本库管理方案,甚至可以帮助用户编写自己的方案。

下一章将会包括一系列建议和最佳实践,其中有具体的 Git 使用技巧,以及和版本控制无关的方法。基于这些建议的某个策略可以被强制性地执行,并且鼓励用户使用本章介绍的工具来提高工作效率。



## 第 12 章

# Git 最佳实践

本书最后一章收集了一些通用的、针对 Git 版本控制的建议和最佳实践。其中很多内容读者在前面章节已经接触过了，这里将会对它们做一个摘要总结。对于每种最佳实践的细节和原因，读者可能还需要参考前面章节的相关内容。

本章涵盖的内容包括工作目录管理，创建单个提交和一系列提交（pull 请求），追加变更提交和变更审核。

本章将会介绍如下主题。

- 如何为软件项目分配版本库。
- 哪些数据需要存储在版本库中，哪些数据应该被 Git 忽略。
- 在创建新提交之前的注意事项。
- 如何创建规范的提交以及一系列提交（换句话说，如何创建一个规范的 pull 请求）。
- 如何选择高效的分支策略，如何给分支和标签命名。
- 如何审核变更，以及如何处理代码审核结果。

### 12.1 启动项目

当启动一个项目时，用户应该选择并明确项目的管理模式（谁管理项目进度，谁集成变更，总之，各司其职）。同时用户还需要决定项目代码的授权许可和著作权利：是否可以用于分发租用；贡献代码是否需要版权转让、贡献者协议、贡献者许可协议，或者只是一个简单的原生数字证书。

## 12.1.1 将工作分配到版本库

在中心式版本控制系统中，几乎所有可以公开的内容都会被放到同一项目树下面。对于 Git 这样的分布式版本控制系统来说，应该将单独的项目分配到独立的版本库中。

最好为每个版库都配置一个概念性的工作组，并事先对其进行正确的划分。如果代码的某个部分涉及多个独立的版本库，那么可以考虑将它单独提取出来成立它自己的项目，后续可以将其作为子模块或者子树与主项目集成。在超级项目中分组的概念可以参考第 9 章的相关内容。

## 12.1.2 选择协作工作流

用户需要确定协作流程，例如项目是否需要使用一个松散的贡献者模型，一个“推举”版本库模型，或者一个中心式版本库等等（详情可以参考第 5 章）。

这经常还需要建立一套访问控制机制，继而确定访问权限结构，详情可以参考第 11 章。同时用户还需要决定如何组织自己的分支，详情可以参考第 6 章。上述决定并不一定要拘泥于教条，随着用户项目和团队经验的不断增长，用户可能会考虑修改分支模型。例如从简单的分支特性模型迁移到纯 Git 流、GitHub 流以及其他衍生模型。

授权许可、协作流程和分支模型的选择应该在开发者文档中明确说明（至少包含 README 和 LICENSE/COPYRIGHT 文件）。用户还需要记住，如果项目开发方式发生了变化，造成这种变化的具体原因是什么。例如，因为项目的发展超越了其最初阶段的形式，这些内容都需要在开发者帮助文档中及时更新。

## 12.1.3 选择需要实行版本控制的文件

大部分情况下，用户不应该将编译生成的文件添加到版本控制系统中（尽管有一些极个别的例外），只跟踪记录源代码（原生的资源文件）。Git 最擅长处理的资源类型是纯文本文件，不过二进制文件也不在话下。

为了避免误将不想被跟踪的文件添加到版本库中，用户应该使用 gitignore 模式。这些专属于项目（例如最终产品的编译系统项目）的忽略模式应该集中放在项目树下的 .gitignore 文件中；专属于开发人员的忽略模式应该被放到其个人的 core.excludesFile 文件（在当前 Git 软件系统中是 ~/.config/git/ignore 文件），或者特定版本库克隆的本地配置文件下，即 .git/info/excludes。详情可以参考第 4 章。

<https://www.gitignore.io> 是新手学习忽略模式的好去处，用户可以根据喜好选择相关的忽略模式。该网站提供了兼容多种操作系统、IDE 和编程语言的.gitignore 模板。

另外一个非常重要的规则是不要将 Git 中会影响系统环境的配置文件添加到版本库中（例如分属于 MS Windows 和 Linux 的配置文件）。

## 12.2 推进项目

下面是一些关于如何创建变更和开发新修订的指导意见。这些指导意见不仅适用于用户个人项目的开发，也适用于用户帮助某个项目维护代码。

不同项目可以采用不同的开发工作流，因此这里提供的某些建议不一定绝对有效，用户需要根据给定项目的开发工作流进行适当调整。

### 12.2.1 使用主题分支

分支在 Git 中的作用主要有两个（详情参考第 6 章）：一是作为维系项目开发人员与特定代码层面的稳定性和成熟性（长期分支）的纽带，另外一个就是作为开发新特性（短期分支）的沙盒。

将变更沙盒化的能力也是将之当作为开发新特性最佳实践的原因。这类分支一般叫作主题分支或者特性分支。独立分支使得开发人员可以方便地在不同任务之间随意切换，并且可以保证多个同时进行的工作任务之间不会互相干扰。

用户最好使用简短并且带有描述性内容的词句给分支命名。主题分支的命名规范有很多种，用户当前项目使用的规范应该在开发文档中明确声明。

一般来说，分支的名称中都会包含其所在主题的摘要描述，其中的单词都是以小写字母的形式出现的，单词之间的空格被连字符或者下划线取代（帮助文档中 `git-check-ref-format(1)` 详细说明了给分支命名时哪些做法是严格禁止的）。分支名可以包含斜杠（用于分层）。

如果用户正在使用一款 bug 管理工具，那么某个修复了 bug 或者解决了问题的分支，可以在其名称前面加上描述问题的工单代码，例如 `1234-doc_spellcheck`。换句话说，维护人员在收集了其他开发人员提交的意见时，可以在主题分支命名过程中将这些意见的首字母放在主题分支名前面，例如 `ad/whitespace-cleanup`（这是一个启发式分支名



的示例)。

用户在某个分支上完成相关工作后,将本地版本库中的分支删除的同时,也要把远程版本库上对应的上游分支一并删除。这是一个非常好的习惯,有助于减少不必要的干扰。

## 12.2.2 确定工作背景

作为一名开发人员,也许经常会被要求在限定的时间之内处理特定的主题分支,例如 **bug** 修复,改进或者修正某个主题或者新特性。

在给定分支上如何开展工作以及基于哪些分支的相关决策信息,取决于项目选定的工作流(详情可以参考第6章)。这类决策还取决于用户的工作类型。

对于主题分支工作流来说(或者主题特性工作流),相对于生成的变更和将要合并的变更来说,开发者将会希望基于最原始和最稳定的长期分支开展后续工作。这是因为,如第6章所述,用户永远不要将一个较不稳定的分支合并到一个较稳定的分支上。此最佳实践原则背后的原因是,不希望因为合并操作带来的种种变化破坏分支的稳定性。

不同类型的变更需要使用不同类型的长期分支上的变更作为基础来研发主题分支。一般来说,为了方便开发者,这些信息应该在开发文档中明确说明;并不是每个人都需要了解该项目使用的分支工作流的所有信息。

接下来根据变更用途的不同,介绍几种常用的基础分支。

- **bugfix**: 这种情况下,主题分支将会基于出现该 **bug** 的历史最久、最稳定的分支创建。通常这意味着从维护性分支上开展工作。如果该 **bug** 没有出现在维护性分支上,那么将会基于稳定分支上的 **bug** 修复分支创建该主题分支。如果 **bug** 并没有出现在稳定分支上,而是出现在了主题分支上,那么相关工作将会基于该主题分支。
- **新特性**: 这种情况下,主题分支(新特性分支)应该是基于稳定性分支创建的。如果新特性依赖的某些特性在稳定性分支上还没有开发完毕,那么可以将工作基于该主题分支进一步展开(根据该主题分支进行构建)。
- **修正和功能改进**: 对于一个还没有达到可以合并到稳定分支要求的主题分支,应该基于该主题分支的外部引用进行功能修正。如果有问题的主题分支还不能发布,那么对该主题分支的修改以及一系列的错误修正都是合理的(详情可以参考第8章历史重写的相关内容)。

如果用户正在工作的项目足够大,有专门的维护人员负责项目的特定部分(子系统),那么首先需要确定的是选择哪个版本库和派生(有时也叫“树目录”)来开展下一步的工作。

## 12.2.3 将变更分解成独立的逻辑单元

除非你的工作非常简单, 只需一个步骤就可以完成一个提交(和大部分 bug 修复类似), 否则用户应该将这些独立的提交分解成独立的逻辑单元变更, 一个提交对应一个单元。这些提交将会被有机地组织起来。

遵循一个良好的习惯编写提交的注释信息(说明用户做了什么), 有助于用户决定何时添加注释。如果用户的注释信息过于冗长, 那么就需要考虑是否是把两个提交压缩在一起了。这一迹象表明用户可能需要将提交分解成更小的几个部分, 并使用更细致的步骤。

不过需要注意的是, 用户还需要考虑的因素包括平衡问题、项目规范、开发工作流的选择等。变更记录至少应该能够表示其自身。对于功能特性开发的每一个步骤(每个提交), 代码编译和程序需要通过测试单元的检验。用户应该及早并经常提交变更。更小的自包含修订记录更易于审核, 更小的但是完整的变更更易于使用 `git bisect` 命令查找遗留的 bug(详情可以参考第 2 章)。

注意, 用户不一定必须循规蹈矩地从头开始执行完美的步骤。当用户发现自己已经疲于应付工作目录下杂乱无章的状态时, 可以使用交互式添加的方式灵活处理(详情可以参考第 3 章和第 4 章)。同时, 用户还可以借助第 8 章讲述的交互式变基或者类似的技术, 在将提交发布之前, 将提交历史整理成易于阅读(并且易于查找定位)的结构。

用户应该谨记, 一个提交是记录用户工作成果(或者指向工作成果的特定步骤)的地方, 而不是保存用户临时工作状态的地方。如果用户需要回退之前临时保存当前的状态, 那么可以使用 `git stash` 命令。

## 12.2.4 编写简洁易读的注释

一个规范的注释应该包含足够的变更细节描述, 以便团队其他开发人员能够评判是否应该将相关代码加入现有代码基中。做出好坏与否的评判应该不需要他们阅读实际的变更, 就能找到提交中引入了哪些新的内容。

提交注释信息的第一行应该尽量简短, 即简洁地(控制在 70 个字以内)描述变更的摘要。如果只包含一段的话, 它应该使用一个空行和其余注释信息隔离开。这么做的具体原因是: 在很多地方, 例如在 `git log --oneline` 命令的输出结果中、`gitk` 这样的图形化历史记录查看器中或者 `git rebase --interactive` 命令的指令表中, 用户只能看到提交注释信息的这一行内容, 并且不得不基于这一行摘要信息来处理相关的提交。

如果用户在提交编写规范的摘要信息时遇到困难，那么这可能需要用户将该提交分解成更小的步骤。

这种变更的摘要行信息存在多种规范。有一种规范是在第一个摘要行中使用 `area:` 做前缀，其中 `area` 代表被修改代码的一般位置：子系统的名称、受影响的子目录或者被修改文件的文件名。如果开发工作是通过 `bug` 追踪系统进行管理的，那么该摘要行可能会是以诸如 `[#1234]` 这样的前缀起头的，1234 是提交中某个任务或者问题的标识符。一般来说，在不能确定该添加什么内容到提交的注释信息时，最好参考开发文档，或者回退到历史上其他提交使用的规范上。



如果用户正在使用敏捷开发方法，那么可以在回顾历史记录过程中找一些比较规范的注释信息，然后将它们作为示例添加到开发文档中以备后用。

对于大部分琐碎的变更，它们的提交注释说明信息可能会需要更长的篇幅才能将意义表述清楚。有一些事情是从其他版本控制系统迁移过来的用户的共识：不允许不为提交添加注释信息或者将所有注释信息都挤在长长的一行中。注意，除非使用了 `--allow-empty` 选项，Git 一般不允许创建一个没有注释信息的提交。

提交注释的信息可能包含的内容如下所示。

- 包含创建提交的理由，解释该提交尝试解决的问题是什么，以及这么做的原因。换句话说，它应该包含当前代码存在的问题或者没有改进之前当前项目行为的描述。这既可以是自包含的，也可以参考诸如 `bug` 追踪系统或者其他外部文档，例如博客、维基条目和统一漏洞披露报告（CVE）。
- 包含一个快速摘要。大部分情况下，它应该解释和评定提交解决问题的效用（为什么）。换句话说，它应该解释用户认为做出这些变更之后的效果要好于之前的原因。这部分的内容不需要解释代码的具体变动，这是代码注释需要完成的任务。
- 如果存在多个解决方案，还需要包括替代性方案最终被放弃的原因，也许还要添加与之有关的讨论和审核意见的链接地址。

确保在不需要查阅任何外部资源的情况下（即不访问 `bug` 追踪系统、互联网或者邮件列表归档），用户就可以读懂开发者编写的注释信息，这是一个比较好的习惯。除了只引用讨论信息或者额外的 URL 地址和问题代码，在提交注释信息中还应编写相关节点的摘要。



编写提交的注释信息时，一个比较推荐的做法是描述变更时使用祈使句，例如让小明去上学，就像用户在给代码基下达命令来改变其行为一样，而不是使用类似“……被修改了”之类的句式。

这里 `commit.template` 和提交注释信息钩子可以帮助用户完成上述最佳实践。详情可以参考第 10 章（以及第 11 章讲述强制性策略的内容）。

## 12.2.5 为提交变更做好准备

在对分支进行变基时，考虑到方便将来的提交，可以将之建立在当前基础分支的外部引用上进行。这样一来将来集成变更时也会更容易一些。如果你的主题分支是基于开发分支或者其他正在研发的主题分支（也许是因为它们依赖于某些特性）构建的，并且该分支的基底分支已经被合并到稳定分支了，那么用户应该使用稳定性集成分支替代上述分支进行变基。

变基的时候也是清理项目历史记录的好时机，这使得用户能够更方便地审核提交的变更记录。只需简单地执行交互式变基，或者根据偏好使用一款补丁管理工具即可（详情可以参考第 8 章）。警告：不要重写已发布的历史记录。

考虑在测试用户提交的变更时对它们进行简洁的合并，如果可能的话，尽量对这些变更进行整理，使得它们更整洁一些，或者条理清楚地将它们合并到集成分支上。

在发送提交之前最好对它们做最后一次复查，以便确保用户对代码的变更中不存在将代码注释掉的情况，并且其中不应该包含与补丁无关的任何额外文件（例如它们不应该包括下一个新特性的变更记录）。在提交代码之前对一系列的提交记录进行审核，以便确保它们的准确性。

## 12.3 集成变更

如何提交已合并变更记录的具体细节和项目使用的开发工作流有关。不同开发工作流的详细介绍可以参考第 5 章。

### 12.3.1 提交和描述变更

如果项目有专门的维护人员，或者至少有专人负责合并相关变更到官方版本库中，那么用户还需要添加提交变更的整体性描述信息（在一系列提交中还要为每个提交添加注释

说明)。在用电子邮件发送变更补丁时，可以通过附件的形式添加。或者可以使用同位贡献者版本库模型时以 pull 请求的注释信息的形式添加。又或者可以在 pull 请求的电子邮件中详细说明，其中在用户公共版本库变更中已经包含了 URL 地址和分支的内容（使用 `git request-pull` 命令生成）。该附件或者 pull 请求应该包含一系列补丁或者 pull 请求用途的说明信息。

考虑为目前正在进行的工作提供一个概述（以及任意相对链接和讨论摘要）；明确说明目前的工作进度，并且在变更描述中进行说明。

在松散的贡献者模型中，被提交的变更通常会以补丁或者系列补丁的形式进行审核。对于邮件列表，如果有可能的话，用户应该使用诸如 `git format-patch` 这样的基于 Git 的工具以及 `git send-email`。多个互相关联的补丁应该被归为一类，例如在用户自己的电子邮件线程中的内容。规范的做法是将它们作为回复邮件的附件一并发送，其中应该包含某个功能特性的整体性描述。

如果变更被发送到了邮件列表中，一个比较常见的做法是在用户的主题行之前加上 `[PATCH]` 或者 `[PATCH m/n]` 这样的前缀（`m` 是一系列 `n` 个补丁的补丁编号）。这使得用户可以方便地将补丁提交与其他电子邮件区分开来。这一任务可以通过 `git format-patch` 命令完成。需要用户自己决定的是否在一系列原生 `PATCH` 名称后面添加额外的标记，例如 `PATCH/RFC`（`RFC` 表示征求意见，即对某个功能实现的创意示例，例如如果该创意有价值的话，那么就应该做一些实验；因为该创意还很幼稚，所以仅限于开发者之间进行讨论验证）。

在同位贡献者版本库模型中，所有开发人员都使用相同的 Git 托管网站或者软件（例如 GitHub、Bitbucket、GitLab，或者它们中的某个私有实例），用户将会把变更推送到自己的公共版本库或者官方版本库的派生上；然后用户会创建一个合并请求或者 pull 请求，这一操作一般是通过托管服务的 Web 接口完成的；然后再次对变更进行整体性的描述。

对于使用中心式版本库的情况（也许还采用了共享维护模型），用户将会在集成版本库中推送变更到一个独立的新分支上，然后向维护人员发出通知，以便其能够找到将要被合并的变更的具体位置。该步骤的细节依赖于精确的设置；发送通知可以通过电子邮件、某种内部消息机制或者通过工单（或者通过 bug 追踪系统中的注释）来实现。

开发文档可能还需要包括发送变更通知的源头和目标地址的规范。它被认为是一种礼节，其目的是为了通知和用户修改变更代码部分有关人员（这里用户可以使用 `git blame` 和 `git shortlog` 命令找到相关人员，详情可以参考第 2 章）。这些人员非常重要，它们可以编写和该变更有关的注释说明，帮助相关人员审核变更代码。

### 人员授权和工作成果签名

为了提高代码跟踪源的效率,某些开源项目会借鉴 Linux 内核名为原生数字证书的签名过程。该签名是提交信息末尾的一个单行信息,例如:

```
Signed-off-by: Random Developer <rdeveloper@company.com>
```



通过添加此行信息,用户可以证明贡献的代码整体或者部分是由该用户创建的,或者基于用户以前的工作,又或者是用户直接提供的内容,任何在这个链条中的人都可以在遵循相应授权协议的情况下根据前人的成果提交变更。如果用户的工作是在某个开发人员工作成果的基础上进行深入拓展的,或者用户只是审核过某人的工作,那么就可以添加多个签名行,形成代码出处链条。

为了给某个辅助创建提交的开发人员授权,用户可以将注释信息末尾追加其他信息,例如 Reported-by:、Reviewed-by:、Acked-by: (这一状态表明,它很有可能涉及代码变更区域的某人) 和 Tested-by:。

## 12.3.2 审核变更的艺术

完成对团队其他开发人员变更的审核工作是非常耗时的(不过这也是使用版本控制的原因),不过好处也是显而易见的:更好的代码质量,减少了代码质量测试和知识迁移所花费的时间。代码变更可以由团队中的其他开发人员进行审核,也可以通过社区审核(需要达成共识),也可以通过维护者或者其助手进行审核。

在进行代码审核过程之前,用户应该通读变更建议的说明,找出建议引入该代码变更的原因,继而确认自己是否是审核该代码变更的最佳人选(这也是规范的注释为何如此重要的原因)。用户需要理解该变更尝试解决的具体问题是什么。用户自身还需要熟悉问题的背景、代码变更发生的具体区域。

第一步是重现代码变更前的问题状态,确认程序是否确实存在相关问题(就像某个 bug 修复提交中对 bug 进行重现一样)。



其次，用户需要签出包含建议变更的主题分支，并验证变更后的成果是否符合预期目标。如果符合，审核相应的代码变更，创建一个包含所有错误的综合列表（不过如果是过程早期的错误，那么也许不必要过于深入追究），如下文所示。

- 提交注释信息描述是否完整？代码是否易读？
- 贡献架构是否正确？体系结构是否合理？
- 代码是否符合项目编码标准和编码约定？
- 代码变更是否仅限于提交注释信息描述的范围？
- 代码是否遵循行业的最佳实践？代码是否安全高效？
- 是否存在任何多余或者重复代码？代码是否最大限度地实现了模块化？
- 代码的测试过程中是否引入了回归测试？如果存在新特性，代码变更是否还包含新特性的测试？是否包含正面和负面两种测试？
- 新增代码执行效率和变更前相比有何差异（项目允许的误差）？
- 所有单词是否拼写正确？新版本内容是否遵循了格式化内容规范？

上述内容只是这类代码审查推荐性检查列表。根据实际项目的具体情况，代码审核过程中可能还有更多的问题需要确认，开发团队还可以编写符合实际需要的检查列表项。用户可以在网上找到很多比较好的示例，例如 Fog Creek 的代码审核列表。

用户在审核代码过程中遇到的问题大致可以分为以下几类。

- **错误：**该特性不在软件项目支持的范围内。有时这类问题还不能被重现。是否其思路已经落后于代码贡献的节奏？如果是这样，那么可以不带偏见地将变更移除，不需要再为此深入分析。
- **不工作：**这通常是代码通过编译之前出现的问题，例如不能通过一系列测试、无法修复 bug 等。这类问题是必须要解决的。
- **失败的最佳实践：**这通常是由于没有遵循行业指导方针或者项目代码规范。相关的代码是否需要微调？这些都是非常重要的调整，不过也可能在选择某种编写方式上存在细微的差别。
- 不符合代码审核者的喜好。提出修改建议，不过不一定必须整改或者澄清。

一些琐碎的问题，例如错别字或者拼写错误，可以由审核人员马上进行修复。如果确实存在一些重复出现的问题，那么可以要求原作者进行修复或者重新提交相关变更。这样

做的目的是为了传播知识。用户最好不要在代码审核过程中做出任何实质性的修改（除非情有可原）。

询问，而不是告知。解释代码应该这样修改的具体原因，提供改进代码质量的方法，区分事实和观点之间的差别。同时要注意它们和在线文档存在的负面偏差。

### 12.3.3 处理审核结果和评论

并不是所有变更在第一次提交时就会被管理员接受。开发者将会收到来自维护人员、代码审核人员和其他开发人员的改进建议。开发者甚至会收到以补丁或者修复提交的形式发来的评论。

首先，怀着一颗感恩的心去面对其他人花时间对你的代码进行审核。如果审核结果存在不清楚的地方，要立即询问当事人进行澄清。如果开发者和审核人员之间存在误会，也需要及时消除。

在这种情况下，接下来的工作就是对代码变更进行优化和打磨，然后用户应该再次提交它们（也许还需要将它们标记为 V2 版本）。开发者应该认真地处理和对待审核人员对变更提出的改进意见。

如果开发者需要处理 pull 请求中的评论，响应方式是类似的。在这种情况下，不同提交是通过电子邮件发送的，开发者可以将注释添加到代码的新版本中（用于回复审核人员或者描述新版本代码和上一版本之间的差异）。既可以在 3 个横杠---之间描述状态差异，也可以在电子邮件的顶部使用“剪刀”行，在提交注释信息说详细加以说明，例如----- >8 -----。变更的解释信息仍然会常驻于迭代记录之间，不过仍然不应该将它们包含到提交注释信息里面；可以通过 `git notes` 命令将它们作为笔记保留（详情可以参考第 8 章），并且可以通过 `git format-patch --notes` 命令自动插入。

根据项目的管理结构，用户应该等待自己提交的变更是否被管理员接受的结果。在开源项目中，这可能取决于项目管理员的独断专行，或者项目带头人、委员会委员或者大家的共识。在提交某个功能特性的最终版本时，最好附加开发者的讨论摘要。

注意，变更在被项目组接受之后，在最终被添加到稳定性分支或者项目中之前，可能仍然需要经过若干考验。

## 12.4 其他注意事项

通过本小节，读者将会学到一些不同于前面章节介绍的最佳实践和指导建议，例如启

动项目、推进项目和集成变更等。

### 12.4.1 不用慌，一切几乎都是可以恢复的

一旦用户提交了自己的工作，将变更存储到版本库中，它们将不会丢失（也许会被用户错过）。Git 还会尝试保留用户当前未提交（保存）的工作，不过不能区分它们，例如因为意外或者主观意愿通过 `git reset --hard` 命令删除工作目录中的变更。为此，用户最好在尝试恢复丢失的提交之前，提交或者暂存用户当前的工作成果。

得益于引用日志机制（对于特定分支和 HEAD 引用），它可以方便地撤销大部分操作。因为其中存放了一系列的暂存变更列表，用户的变更可能就藏身其中。还有就是 `git fsck` 命令可以作为最后的救命稻草。关于数据恢复的细节可以参考第 11 章。

如果用户的工作目录的状态是一团乱麻，那么请停下来想一想。不要轻易地丢弃自己的工作成果。在交互式添加、交互式重置（以及 `--patch` 选项）以及交互式签出等操作的帮助下，用户通常是可以将杂乱无章的工作目录打理得井井有条的。

执行 `git status` 命令，然后仔细地阅读其输出结果，大部分情况下将会有助于使用户从某些比较少见的 Git 操作中中断中脱困。

如果用户在变基或者合并操作方面遇到问题，那么不应该不负责任地将它们扔给其他开发人员，因为总有第三方的 `git-imerge` 工具作为解决方案。

### 12.4.2 不要修改已发布的历史记录

一旦用户发布了自己的变更，那么用户最好认为这些修订像石头一样不可变、不可编辑。如果用户发现提交中存在问题，那么可以创建一个修复提交来应对（也许还需要用到能够撤销变更影响的 `git revert` 命令），与之有关的详情可以参考第 8 章。

除非在开发文档中明确说明这些特定分支可以被重写或者重做，否则最好还是避免创建这类分支。

在极个别情况下，用户可能确实需要修改历史记录：例如移除一个文件，清理未加密的密码信息，移除误操作而添加的大型文件等等。如果用户需要这么做，那么最好事先通知开发团队中的所有成员。

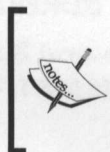
### 12.4.3 版本发布的数字化和标签化

在用户发布项目新版本之前，可以使用一个签名标签对新版本进行标记。这确保了刚



创建的修订记录的完整性。

在为软件标签和软件版本号进行命名时有多种规范。最常见的一种是使用标签化版本号，例如使用 1.0.2 或者 v1.0.2 作为标签名。



如果集成工作对于项目来说非常重要，可以考虑为集成使用签名合并（即合并签名标签），详情可以参考第 5 章和第 11 章。

在软件新版本命名上存在多种不同的规范，例如使用基于时间的版本。有一种约定是在软件版本名称后添加时间，例如 2015.04（或者 15.04）。其次，还有一个比较常见的语义化版本约定（<http://semver.org/>），一般是通过 MAJOR.MINOR.PATCH 的形式进行数字化表示的，当用户为了向下兼容添加了 bug 修复提交时，可以增加 PATCH 的数值，MINOR 的数值增加表示为了添加某个软件功能的向下兼容性，MAJOR 的数值增加表示新增了不兼容 API 的变更。即使不使用一个完整的语义化版本名称，一个比较常见的做法是为维护性的版本添加第三个数字，例如 v1.0 和 v1.0.3。

## 12.4.4 尽可能自动化

用户不仅需要将代码规范写在开发文档中，而且还需要强制遵守该规范。遵循这些规范可以方便系统调用客户端钩子（详情可以参考第 10 章）。可以使用服务端钩子强制执行上述规范（详情可以参考第 11 章）。

钩子可以在 bug 追踪系统中帮助用户自动管理工单，在提交注释信息中基于给定触发器（模式）执行特定的操作。钩子还可以用于防止重写历史记录。

可以考虑使用第三方的解决方案，例如 Gitolite 或者 GitLab，以便加强强制性规则的访问控制。如果用户需要对代码进行审核，相应的工具有 Gerrit、GitHub 的 pull 请求、Bitbucket 和 GitLab。

## 12.5 小结

这些建议都是基于将 Git 作为版本控制系统的最佳实践而提出的，能够切实促进用户的开发工作和团队建设。用户的学习之路又上了一个台阶，从一个想法开始，最后以变更集成到项目中结束。这些检查列表可以帮助用户开发出更好的代码。

# Git高手之路

Git是常用的源代码管理（Source Code Management, SCM）和分布式版本控制系统（Distributed Version Control System, DVCS）之一。了解Git背后的理念和架构有助于读者深入挖掘它的潜力并理解其行为。学习最佳实践和推荐的工作流模型，让用户的程序开发工作如虎添翼。

本书是经过精心编写的，旨在帮助读者深入理解Git架构，以及其内部的理念、行为和最佳实践。除此之外，本书还有助于读者掌握诸多实用技能，例如检查和浏览项目历史记录，创建和管理工作成果，在中心式和分布式工作流中配置版本库和分支以便协作开发、集成来自其他开发人员的工作成果，自定义和扩展Git，以及版本库数据恢复等。通过了解Git高级应用技巧和内部工作机制，读者将会深入理解Git的行为，从而能够自定义和扩展现有的工具和脚本，甚至编写符合自己需要的功能。

## 本书的目标读者

如果你已经是一名Git用户，并且掌握了分支、合并、暂存和工作流等基本概念，那么本书是为你而写的。在阅读本书之前，读者最好先了解安装Git的基本知识和软件配置管理的概念。

## 阅读本书，你将从中学到：

- 浏览项目历史记录，使用不同条件查询修订记录，过滤和格式化项目历史记录；
- 管理工作区和暂存区，以及交互式创建和编辑新的修订；
- 为团队协作配置版本库和分支；
- 通过合并和变基操作，提交自己的工作成果并集成他人的工作成果；
- 自定义Git的系统级行为，其影响范围可以是单个用户、单个版本库和单个文件；
- Git版本库的配置和管理、访问控制、版本库数据恢复和日常维护；
- 选择工作流模型，并配置与之兼容的运行环境。



异步社区 [www.epubit.com.cn](http://www.epubit.com.cn)  
新浪微博 @人邮异步社区  
投稿/反馈邮箱 [contact@epubit.com.cn](mailto:contact@epubit.com.cn)

美术编辑：董志桢

分类建议：计算机 / 软件开发  
人民邮电出版社网址：[www.ptpress.com.cn](http://www.ptpress.com.cn)

ISBN 978-7-115-47850-4



9 787115 478504 >

ISBN 978-7-115-47850-4

定价：89.00 元